

DISKETTE IM HEFT

64'er

ASSEMBLER

LEICHT GEMACHT

Intensivkurs

Von Basic zur Maschinensprache

Wandposter

Alle Befehle auf einen Blick

Lösungen

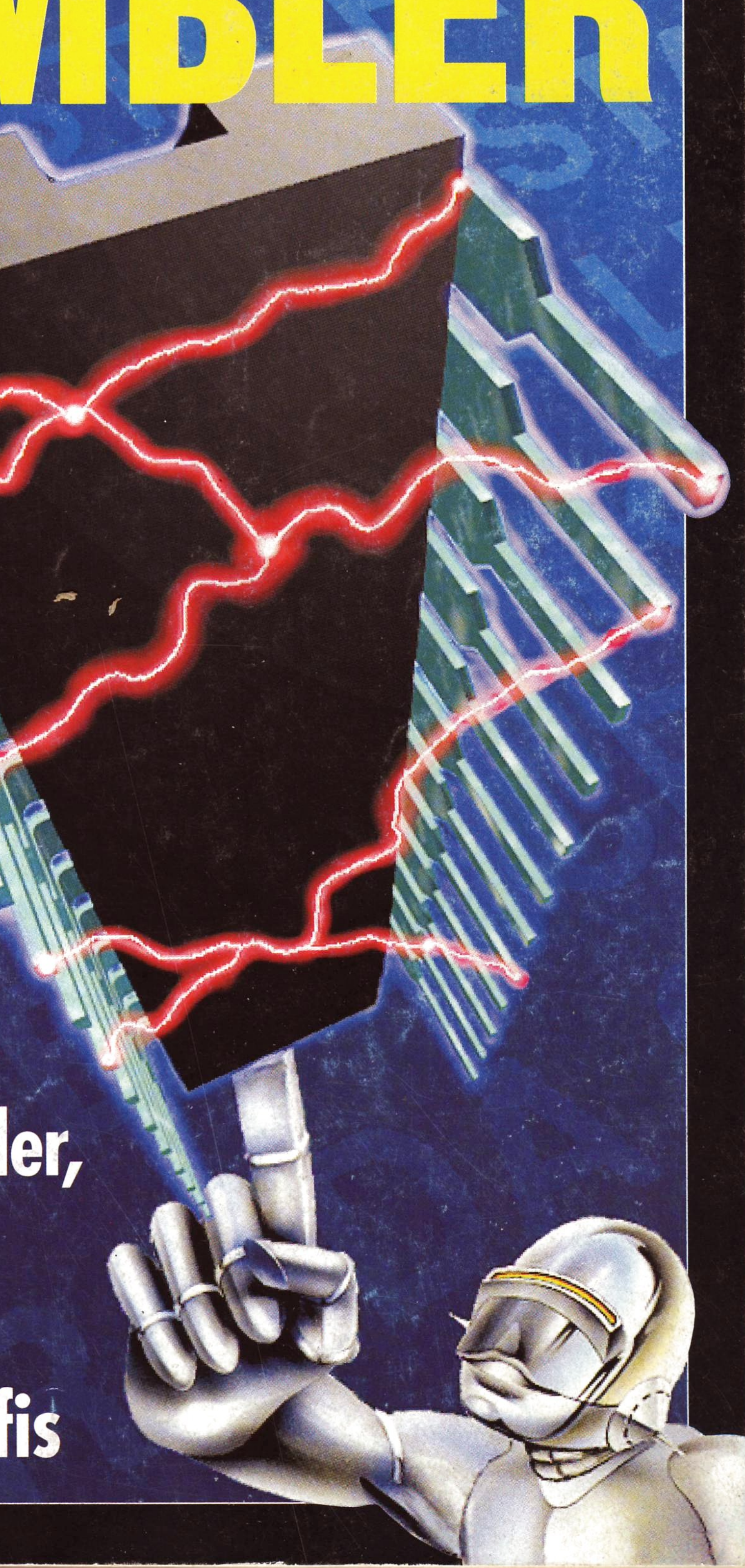
Assembler-Programmierung

Komplettpaket

Assembler, Reassembler, Speichermonitor

Tips & Tricks

Für Anfänger und Profis



64'er

über 50 Programme auf Diskette

Top-Spiele



Bitte Coupon ausfüllen, auf frankierte Postkarte kleben und senden an: 64er Leserservice, CSJ Postfach 140 220, 8000 München 5 oder telefonisch unter 089/ 20 25 15 28 bestellen.

für 9,80DM..

64'er

Hiermit bestelle ich
___ Exemplare 64er Sonderheft Top-Spiele

Zum Einzelpreis von 9,80DM (zzgl. 3,-DM Versandkosten). Die Bezahlung erfolgt nach Erhalt der Rechnung.

____
Name, Vorname

____
Straße, Hausnummer

____
PLZ, Wohnort

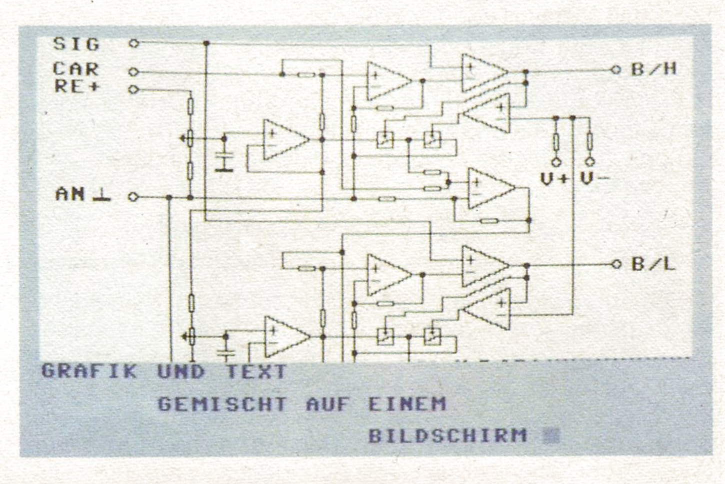
AC 26 18

Telefon (Vorwahl) ______________________

Hiermit erlaube ich Ihnen mir interessante Zeitschriftenangebote telefonisch zu unterbreiten (ggfs. streichen).







Geteilter Bildschirm
Hires und Text zusammen bietet »Splitscreen«
Seite 49



Mehr als 16 Farben
Wer hätte das gedacht – 136 Farben am C64
Seite 45

Kurs

Poster

Tools

Tips & Tricks

Sonstiges

Alle Programme zu Artikeln mit einem ■-Symbol finden Sie auf der beiliegenden Diskette (Seite 19).

Assembler-Kurs – hinter den Kulissen



Der C64 bietet Grafik, Sound und vieles mehr. Doch für denjenigen, der alles ausprobieren möchte, ist die erste Zeit hart. Aus dem (etwas kümmerlichen) Handbuch werden die Basic-Beispiele mühsam in die Tastatur geklopft. Irgendwann kommen (Gott sei Dank!) die ersten Erfolgserlebnisse. Wer allerdings nach den ersten durchprogrammierten Nächten festgestellt hat, daß zwar alles funktioniert – aber unendlich langsam – steht vor der Alternative: Ich werde reiner Anwender oder versuche es mal mit Maschinensprache. Für den leichteren Weg gibt es eine Unzahl von Programmen. So viele, daß allein mit den Titeln ein Buch gefüllt werden könnte. Um hier zum Erfolg zu kommen, hilft nur suchen, kaufen, tauschen.

Für alle anderen beginnt ein harter Kampf nach Grundinformationen, Tips und Hilfestellungen – und genau für diese Gruppe zukünftiger Fachleute ist dieser Kurs gedacht.

Was ist Maschinensprache?

Beginnen wir unseren Excurs mit der Anwendersprache Basic, genau so, wie sich unser C64 nach dem Einschalten meldet. Halt! Diese Behauptung stimmt nicht, denn bis zu dem Augenblick, da Sie die Einschaltmeldung am Bildschirm sehen, hat der Computer schon eine Unmenge von Maschinenprogrammen abgearbeitet. Beispielsweise wurde der Bildschirm-Chip initialisiert, alle Sound-Parameter zurückgesetzt und der Speicher auf die (vielleicht) folgende Arbeit vorbereitet. Und das ist noch lange nicht alles: auch jetzt ist er damit beschäftigt, den Cursor blinken zu lassen und wartet auf Ihre Eingaben. Selbst wenn Sie jetzt die berühmte Befehlsfolge

```
PRINT"HALLO"
```

eingeben, und diesen Text mit <RETURN> bestätigen, beginnt er wieder eine Reihe Maschinenprogramme abzuarbeiten. Zuerst wird überprüft, ob die Schreibweise von »PRINT« einem Befehl entspricht, dann werden die darauffolgenden Zeichen eingelesen, gecheckt usw. Danach erscheint – nach langer Arbeit für den C64, aber in Sekundenbruchteilen für uns – der Text »HALLO« am Bildschirm. Die Unzahl von Maschinenprogrammen, die unser Computer gerade abgeschlossen hat, nennen wir »Basic«. Diese Programmiersprache ist ein ausgeklügeltes System, das alle Eingabefehler kommentiert, ja zum Teil verhindert und uns komfortable Befehle zur Verfügung stellt. Alle Systemroutinen bestehen aus einer Folge von Zahlen, die der Reihe nach abgearbeitet werden.

Irgendwer (Microsoft) hat irgendwann (1976) einmal das Grundkonzept von Basic entworfen. Und da es ziemlich aufwendig ist, aus einer Tabelle von Befehlen entsprechende Zahlen herauszusuchen und diese dann einzeln in den Speicher zu bringen, verwendete man ein Hilfsprogramm, den Assembler. Er stellt (wieder) ein Maschinenprogramm zur Verfügung, welches leicht zu merkende (3buchstabige) Mnemonics in einen vom Mikroprozessor direkt verwendbaren Code umwandelt. Diese Bytes werden ab einer vom Programmierer bestimmten Stelle im Speicher abgelegt.

Übrigens ist der Mikroprozessor nicht das einzige Bauelement unseres C64. Es gibt zusätzlich: den Video-Chip (VIC), einen Sound-Chip (SID), eine Menge Schaltkreise, die alles verknüpfen und natürlich die 64 K-Byte Schreib-Lese-Speicher, von denen unser Computer seinen Namen hat. Sie alle stehen aber unter dem Kommando des Mikroprozessors.

Wie funktioniert der Mikroprozessor?

Sie haben es vorhin schon gehört: Ein Computer verarbeitet Zahlen. Das ist so, auch wenn Sie am Bildschirm Grafiken bewundern, der Printer Texte druckt, und der Lautsprecher Töne von sich gibt. Alle Tätigkeiten finden ihren Ursprung in einer Anreihung von Zahlen im Speicher, die der Mikroprozessor analysiert und die ihm sagen, was er zu tun hat – und (sehr wichtig!) wie viele der nachfolgenden Speicherstellen er dazu benötigt. Gehen wir von einem unverrückbar festen Zustand aus: dem RESET. Immer wenn Sie den C64 einschalten oder den RESET-Knopf (falls vorhanden) drücken, beginnt der Mikroprozessor zu arbeiten. Er springt an eine definierte Stelle im Speicher und liest den ersten Zahlenwert. Man kann diese erste Zahl auch als Befehl bezeichnen. Er legt fest, wie viele der nachfolgenden Speicherstellen zur Befehlsausführung nötig sind. Es können null bis zwei Byte sein.

Was ist ein Absturz

Damit wird auch klar, was passiert, wenn Sie willkürlich an eine Stelle im Programm springen (z.B. mit »SYS64738« aus Basic). Erwischt der Mikroprozessor eine Stelle, an der ein zufälliger Wert, aber kein Befehl steht, versucht er diese Zahl als Anweisung zu interpretieren. Das funktioniert natürlich nicht, da die nachfolgenden Stellen nichts mit diesem vermeintlichen Befehl zu tun haben, also unserem Mikroprozessor falsch einsagen. Meistens (so auch bei unserem Beispiel) führt dies zum Absturz des Systems (d.h. unser C64 läßt sich erst wieder durch RESET oder Ein- und Ausschalten zum Leben erwecken). Manchmal passieren auch seltsame Dinge: z.B. die Bildschirmfarben ändern sich oder der Cursor blinkt schneller usw. Diese Reaktion kann übrigens auch bei einem Programmierfehler auftauchen. Sie ist der unangenehmste Unterschied zu einer Hochsprache: Basic fängt alle falschen Eingaben ab, steigt mit einer Fehlermeldung aus dem laufenden Programm und der Programmierer kann anhand der Fehlermeldung seine falsche Eingabe korrigieren – danach versucht er's halt nochmal.

Ein Maschinenprogramm dagegen stürzt bei gravierenden Fehlern ab, d.h. der Computer reagiert einfach nicht mehr. Um den Fehler zu analysieren, muß er zuerst aus seinem Tiefschlaf gerüttelt werden, erst danach läßt sich das Programm überprüfen. Doch wie hole ich den Computer wieder ins Leben zurück, wenn er hängt?

Wasser

Ich programmiere in Maschinensprache! »Respekt«, sagt man beeindruckt. Warum eigentlich? Assembler setzt keine besondere Intelligenz voraus. Nur kennt und hütet sein Anwender ein paar Geheimnisse. Diesen Schleier lüften wir für Sie.

1. Es gibt die brutale Methode: aus- und wieder einschalten. Sie funktioniert zwar immer, aber hat den Nachteil, daß unser Programm (falls überhaupt gespeichert) wieder neu geladen werden muß.

2. Die elegantere Methode ist ein RESET-Taster. Er wurde zwar vom Hersteller weggelassen, läßt sich aber leicht nachrüsten (Sonderheft 57, Seite 44). Drei Möglichkeiten bestehen für die Installation: Extern am User-Port, Extern am seriellen Ausgang, oder intern eingebaut. So ein Taster gaukelt dem C64 vor: "Du bist gerade eingeschaltet worden". Und tatsächlich, der Mikroprozessor beginnt wieder zu leben. Im Unterschied zum Ausschalten ist aber der Speicher nicht gelöscht (wenn es auch zunächst so aussieht). Die Programme sind alle noch vorhanden und lassen sich mit ein bißchen Know-how zurückholen (Sonderheft 57, Seite 24 »Reset ohne Reue«).

Begriffserklärungen

Byte

Die Behauptung, ein Computer könne nur Zahlen verarbeiten, stimmt nicht ganz, denn er kennt eigentlich nur zwei Zustände: Strom eingeschaltet oder nicht. Da es für alles eine Bezeichnung gibt, nennen wir dieses Geschehen 1 Bit. Damit allein läßt sich natürlich nicht viel anfangen. Denken Sie an Ihre Nachttischlampe: Sie ist entweder eingeschaltet oder nicht, und damit sehen Sie entweder etwas oder nicht. Um mehr Möglichkeiten zu erreichen, hat man mehrere dieser Bits zusammengefaßt. Denken Sie an zwei Nachttischlampen: Entweder Sie sehen etwas, oder Ihr Bettnachbar, oder beide. Beim (binären) Zählen steht jeder Möglichkeit eine Zahl gegenüber. Nehmen wir als Beispiel die Kombinationen von 4 Bit. Mit ihnen lassen sich 16 Variationen (Zahlen) darstellen. Welche davon einem Zahlenwert entspricht, wurde folgendermaßen festgelegt:

0 = 0000
1 = 0001
2 = 0010
3 = 0011
4 = 0100
5 = 0101
6 = 0110
7 = 0111
8 = 1000
9 = 1001
10 = 1010
11 = 1011
12 = 1100
13 = 1101
14 = 1110
15 = 1111

Sie sehen hier wird ein festes System verwendet. Fügt man ein Bit hinzu, verdoppelt sich jeweils die Anzahl. Unser C64 faßt genau 8 Bit zusammen (und bearbeitet sie auch gleichzeitig). Damit ergeben sich 256 Kombinationen. Da man diese Zusammenfassung ein Byte nennt, läßt sich pro Byte eine Wertereihe von 0 bis 255 darstellen.

Mnemonic

... ist lt. Fremdwörterlexikon ein Mittel, um das Gedächtnis durch Merk- oder Lernhilfsmittel zu unterstützen, die mit dem zu Merkenden in äußerliche Verbindung gebracht werden können. In verständlicherem Deutsch sind dies (mehr oder weniger) verständliche Abkürzungen für Tätigkeiten, die der Rechner erledigen soll. Beispielsweise bedeutet »LDA«: Lade den Akku (das Hauptregister s.unten).

Mikroprozessor (CPU)

... ist der wichtigste Bestandteil eines Heimcomputers. Und in der Tat ist dieses Bauteil ein (mikroskopisch kleiner, ca. 5 x 6 mm großer) integrierter Schaltkreis (IC) mit der Bezeichnung »6510«, der alle anderen elektronischen Bauteile Ihres C64 steuert und zusätzlich noch eigenständige Berechnungen durchführt. Damit man ihn überhaupt im Rechner einlöten kann, machte man ihn künstlich größer – er wurde in einen Plastikmantel eingegossen. Ohne den Mikroprozessor könnte unser C64 nicht einmal den Cursor am Bildschirm darstellen. Der 6510 ist ein Nachfolgetyp des bei Commodore häufig verwendeten 6502 (2001 PET, 30xx-Serie, Floppystationen usw.). Die Befehlssätze beider Mikroprozessoren sind identisch. Der größte Unterschied zu seinen Vorgängern ist ein eingebauter Port (sechs herausgeführte, frei programmierbare Leitungen). Dieser steuert im C64 die Kassettenfunktionen und die Speicherverwaltung, denn der C64 verwaltet mehr als 64 KByte Speicher. Aber dazu kommen wir später.

Warum braucht man einen Assembler?

Wie wir schon mehrmals gehört haben, besteht ein Maschinenprogramm aus Zahlenkolonnen unterschiedlicher Länge. Für die Kombination von Befehlen und den nachfolgenden Bytes gibt es schier unzählig viele Kombinationen (s. Poster S. 26/27). Also ist die Methode: Befehl aus der Tabelle heraussuchen, in die Speicherstelle POKEn, die Anzahl der nötigen nachfolgenden Zahlen suchen und danach deren Werte in die nächsten Speicherstellen POKEn – nicht gerade effektiv. Das Hilfsprogramm »Assembler« nimmt uns diese Arbeit ab. Es übersetzt für uns verständliche Bezeichnungen (Mnemonics) in Zahlen (Befehlscode) für den Mikroprozessor, berechnet die Anzahl der nachfolgenden Speicherstellen und füllt sie mit den richtigen Werten. Zusätzlich wird beim Assemblieren (Übersetzen in Maschinencode) ein Protokoll ausgegeben. Leichte Fehler werden sogar angezeigt. Zusätzlich läßt sich der Quelltext auf Diskette speichern, später wieder laden und umarbeiten. Damit entfällt auch ein Nachteil der direkten Maschinenprogrammierung: bei jeder noch so kleinen Änderung muß (durch unterschiedliche Befehlslängen bedingt) alles neu berechnet und eingetragen werden. Der Assembler macht's automatisch.

Woraus besteht der Mikroprozessor?

Das ist ein so komplexes Thema, daß wir damit mehr als ein Buch füllen könnten. Lassen wir darum komplexe Details weg:

Zunächst ist der Mikroprozessor ein eigenständiges Bauteil, das intern aus mehreren einzelnen Baugruppen besteht. Damit diese auch wirklich Hand in Hand arbeiten, benötigt die CPU einen Arbeitstakt (Clock). Man kann ihn sich wie ein Pendel vorstellen, das den Prozessor für jeden Arbeitsprozeß einmal anstößt. Im Gegensatz zu einem Uhren-Pendel, bewegt sich das Clock-Pendel sehr schnell, nämlich 970000 mal pro Sekunde.

Im 6510 existiert für jede Aufgabe eine einzelne Baugruppe:

1. Die arithmetische Recheneinheit (ALU – Arithmetic Logic Unit) – ist für Organisation und Rechenoperationen zuständig.

2. Das Hauptregister (Akku) – normalerweise wird hier ein Byte geladen, bearbeitet und wieder zurück in den Speicher geschrieben. Wir werden später sehen, daß einige Operationen auch mit anderen Registern möglich sind.

3. x-Register – ein Hilfsregister, das eingeschränkte Rechenoperationen beherrscht.

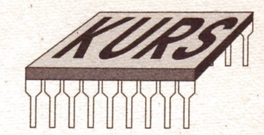
4. y-Register – das zweite Hilfsregister, auch mit ihm sind eingeschränkte Rechnungen möglich.

5. Statusregister – spiegelt die Reaktionen auf Rechenoperationen wider. Man sagt auch »zeigt den Prozessorstatus an«.

6. zusätzliche Hilfsregister (Status und Stapel)

7. der Programmzähler (PC – hier: Programm-Counter)

Alle Bestandteile, mit der Ausnahme des PCs, können nur mit einem Byte umgehen. Der Programm-Counter wird zwar mit 16 Bit angesprochen, besteht aber aus zwei 8-Bit-Registern (PCL, PCH). Der Grund dafür ist, daß er alle Speicherzellen anwählen und erreichen muß. 16 Bit ergeben 65536 Möglichkeiten, also genau die Anzahl, die unser C64 an Speicherzellen verwaltet. Zum Unterschied zu dieser elektronischen Baugruppeneinteilung gibt es die für Sie bedeutend wichtigere Gliederung in programmierbare Einheiten. Im Textkasten »Programmierbare Bestandteile des 6510« sind



Programmierbare Bestandteile des 6510

Programmzähler (PC)

... sein Inhalt bestimmt, aus welcher Stelle im Speicher der Mikroprozessor die nächsten Daten liest. Vereinbart ist, daß der erste gelesene Wert einem Befehl entspricht. Dieser Zahlenwert (Befehl) bestimmt, was getan werden soll und wieviel Speicherzellen zur Ausführung benötigt werden. Von diesem Zahlenwert hängt es also ab, welche Register angesprochen werden, ob der Programmzähler selbst manipuliert wird (z.B. durch eine Reaktion auf ein Rechenergebnis) oder eine Rechnung mit dem Hauptregister durchgeführt wird. Der Programmzähler kann Werte von 0 bis 65535 verarbeiten und damit jede Stelle des Speichers erreichen. Befehle, die den Programmcounter ändern, sind:

JMP, JMP ($$$$), RTS, RTI, BNE, BEQ, BPL, BMI, BVC, BVS, BCC, BCS, BRK und NOP

Hauptregister (Akku)

... stellt das wichtigste Register dar. Mit ihm läßt sich aus Speicherstellen laden, addieren, subtrahieren, logisch mit einer Speicherstelle oder einem Wert verknüpfen und das Ergebnis in eine Speicherstelle oder in das x-, y-Register übertragen. Ebenso läßt sich ein Wert direkt aus den x-/y-Registern in den Akku übertragen. Der Akku kann nur Werte von 0 bis 255 verarbeiten. Befehle für den Akku:
LDA, STA, TAX, TXA, TAY, TYA, ASL, ROL, LSR, ROR, AND, EOR, ORA, BIT, ADC, SBC, CMP, PLA, PHA, PHP und PLP

x-Register

... ist das erste Hilfsregister, kann aufwärts und abwärts zählen und Additionen bzw. Subtraktionen durchführen. Zusätzlich hat es eine wichtige Aufgabe: sein Inhalt läßt sich mit einer Reihe von Befehlen mit dem Akku oder einer Speicherstelle verknüpfen und deutet dann auf Speicherstelle+x (x-indizierte Adressierung). Auch hier ist nur ein Wertebereich von 0 bis 255 erlaubt. Befehle, die direkt das x-Register betreffen:
LDX, STX, INX, DEX, TSX und TXS.

Befehle, bei denen das x-Register beteiligt ist:
LDA $$$$,X, LDY $$$$,X und STA $$$$,X

y-Register

... ist das zweite Hilfsregister und kann das meiste des x-Registers (Ausnahme TSX und TXS). Zusätzlich hat es eine Erweiterung der Indizierung. Zusammen mit dem Akku kann es jeweils zwei Speicherstellen der Zero-Page (= Speicherbereich < 255) als Zeiger (Pointer) verwenden. Das Interessante dabei ist: nicht der Wert aus diesen Speicherstellen wird verwendet, sondern die Adresse aus den Zellen, plus den y-Wert. Wie das genau funktioniert, werden wir noch kennenlernen. Das y-Register verkraftet Werte von 0 bis 255.

Befehle, die direkt das y-Register betreffen:
LDY, STY, INY und DEY

Befehle, bei denen das y-Register beteiligt ist:
LDA $$$$,Y, LDX $$$$,Y, STA $$$$,Y,LDA ($$$$),Y und STA ($$$$),Y

Statusregister

...zeigt die Reaktionen bei Rechenoperationen an, z.B. ein negatives Ergebnis, ein Überfließen oder ein Ergebnis = 0. Dabei hat jedes einzelne Bit eine eigene Funktion. Sie werden sich sicher gefragt haben, wie man größere Zahlen als 255 bearbeitet, wenn im Akku und den einzelnen Registern nur Werte einschließlich 255 erlaubt sind. Das Statusregister gibt Auskunft darüber, ob bei einer Rechnung das Ergebnis kleiner 0, gleich 0 ist, oder etwa der Bereich (255) überschritten wurde. Anhand dieses Zustands kann (muß aber nicht) auf einzelne Ereignisse reagiert werden. Um aber für Rechenoperationen die Voraussetzungen zu schaffen, lassen sich einzelne Bits direkt beeinflussen. Zusätzlich läßt sich durch Setzen eines Bits der IRQ sperren (SEI) oder durch Löschen (CLI) wieder freigeben. Ein neuer Begriff – IRQ. Sie haben sicher schon einmal davon gehört. Jede 60stel-Sekunde unterbricht der Mikroprozessor sein Programm, merkt sich den letzten Status, und führt (bei Basic) die Tastaturabfrage durch. Dieser Zustand läßt sich im Wortschatz des Mikroprozessors mit einem speziellen Befehl ab- und wieder einschalten. Damit wird das entsprechende Bit im Status-Register gesetzt. Befehle, die einzelne Bits des Status-Registers beeinflussen:

SEI, CLI, CLC, SEC, CLV, CLD und SED.

Stack

... ist der Zwischenspeicherbereich (Stapel) des Mikroprozessors. Die Bezeichnung »Stapel« ist bezeichnend für das Prinzip der Speicherung: »last in, first out« (das zuletzt gestapelte muß als erstes wieder hinaus). Wir alle kennen dieses Prinzip. Stellen Sie sich einen Stapel Papier vor. Das zuletzt abgelegte Papier muß als erstes wieder entnommen werden, um an die unteren zu kommen. Genau nach diesem Prinzip funktioniert der Prozessor-Stack (Adresse 256 bis 511). Interessant ist, daß Sie sich um die Verwaltung des Stapels nicht kümmern müssen: er wird automatisch vom Prozessor verwaltet.

Beispiel: Die CPU merkt sich beim IRQ die Position, an der das Programm unterbrochen wurde (zusätzlich noch den Status). Nach Beenden des Interrups holt er sich diese Daten wieder. Um sie aber zu finden, muß er sich merken, an welcher Stelle im Stack er sich gerade befindet, denn darunter befinden sich die vorher abgelegten Daten. Dafür ist der Stack-Pointer zuständig. Sein Inhalt (0 bis 255) kann durch einige direkte Befehle verändert werden. Doch Vorsicht: ein unkontrolliertes Manipulieren des Stacks führt fast immer zum Absturz des Mikroprozessors:

PHA und PLA.

diese Einheiten beschrieben. Zusätzlich erhalten Sie hier eine Auflistung der Mnemonics für die betreffenden Befehle. Eine Erklärung folgt später.

Die Philosophie von Maschinensprache

In Basic bewirkt ein Befehl, je nachdem unter welchen Voraussetzungen Sie in verwendet haben, eine Reihe von Reaktionen. Nehmen wir als Beispiel »PRINT«: Dieser Befehl gibt einen Text auf den Bildschirm, auf den Drucker oder sogar auf die Floppy aus. So komplexe Anweisungen gibt es in Maschinensprache nicht. Hier begeben Sie sich zu den Wurzeln des C 64. Jeder von 56 Befehlen ruft eine ganz bestimmte Reaktion des Computers hervor. Daß eine Reihenfolge dieser Maschinenbefehle aber wieder komplexe Reaktionen hervorrufen können, sieht man anhand von Basic. Es besteht aus vielen unterschiedlichen Maschinenprogrammen, die sich gegenseitig ergänzen. Hier schließt sich der Kreis. Der Ursprung aller Programmiersprachen sind Maschinenprogramme, die sich selbst überprüfen und zum Teil sehr komplexe Aufgaben erfüllen. Wie das möglich ist, wollen wir uns anhand eines Beispiels ansehen:

```
POKE 1024,13
```

Wie Sie wissen, bringt dieser Befehl in die Speicherstelle 1024 den Wert 13. Das ist nichts Außergewöhnliches, werden Sie sagen. Aber zufällig ist »1024« der Beginn des Bildschirmspeichers (Ende = 2024). Er ist ein Teil des normalen Speichers des C 64 – mit einem Unterschied: Von Zeit zu Zeit (jede 50stel Sekunde) liest der Video-Interface-Chip (VIC) diesen Speicherbereich und schaut nach, welche Werte hier vorhanden sind. Nach (für uns) nicht spürbarer Zeit hat er ein Muster aus einem anderen Speicherbereich herausgesucht, das diesem Wert entspricht und ein »M« links oben dargestellt.

»PRINT« kann das auch, sogar noch mehr. Löschen Sie doch mal den Bildschirm mit <SHIFT CLR/HOME>, fahren Sie mit dem Cursor ein paar Zeilen tiefer und geben Sie ein:

```
PRINT CHR$(19)"M"
```

Nach <RETURN> erscheint »M« wieder an der gleichen Stelle wie vorher.

Der gravierende Unterschied zwischen beiden Methoden ist die Philosophie, die dahintersteckt. Beim POKEn wurde ein Maschinenbefehl simuliert. Wir benötigten die Kenntnis, wo der Bildschirmspeicher beginnt und in welche Speicherstelle wir welchen Wert POKEn mußten, um das »M« am richtigen Platz sichtbar zu machen. Beim PRINT mußten wir nur angeben: gehe an die obere, linke Bildschirmposition (CHR$(19)) und stelle dort ein »M« dar ("M").

Sie sehen den Unterschied zwischen Maschine und Basic. Bei Basic gibt es umfassende Befehle, die Ihnen Denkarbeit abnehmen, bei Maschine benötigen Sie genauere Kenntnis, was im Computer vorgeht. Natürlich gibt es Programmierer, die alle diese Dinge auswendig beherrschen, aber das sind die wenigsten. Für Sie empfehlen wir ein paar Grundlagenwerke und -artikel, in denen beschrieben wird, welche Speicherstellen für welchen Zweck reserviert sind. Denn beim C 64 sind alle Reaktionen durch Schreiben (in Basic POKEn) und Lesen (PEEK) in/aus Speicherstellen möglich. Man muß nur wissen wo. Dazu einige Literaturverweise zu unseren 64'er Sonderheften (Preis: 16 Mark inkl. Diskette):

»Sprites«, SH 62, S.34
»Zero-Page« SH 65, S.24
»Port-Bausteine« SH 65, S. 30
»VIC+Interrupt« SH 65, S. 34
Markt & Technik Leserservice, CSJ, Postfach 1 40 20, 8000 München 5, Tel. 089/20251528

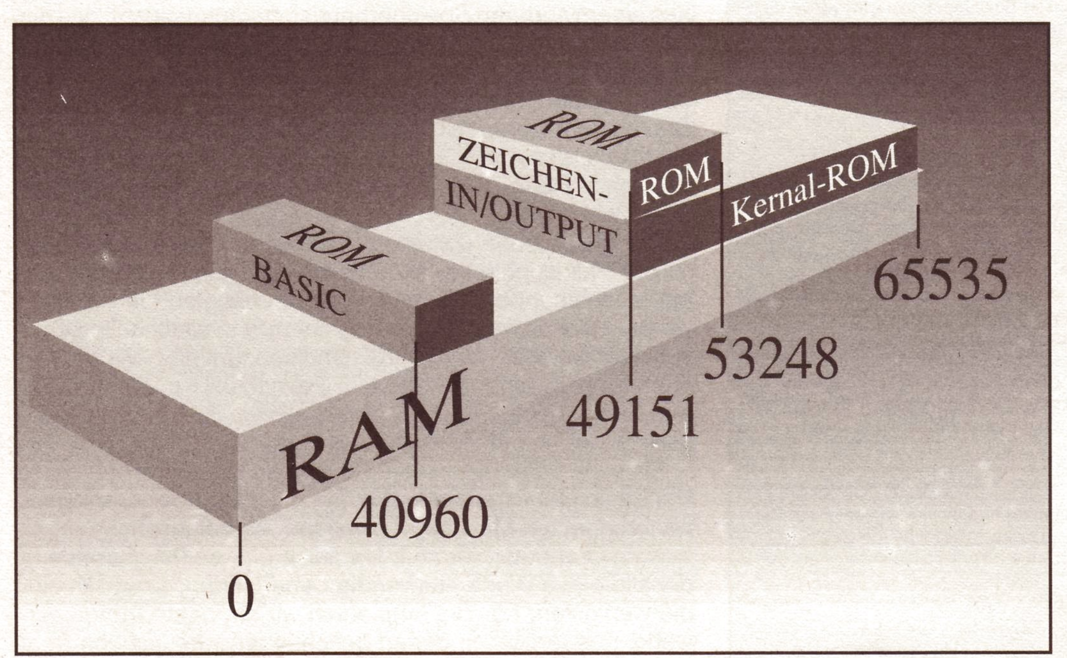
Zusätzlich empfehlen wir Ihnen »das« Standardwerk für den Assembler-Programmierer:

Brückmann/Englisch/Felt/Gelfand/Gerits/Krsnik, Das neue Commodore-64-Intern-Buch, 836 Seiten, 29,80 Mark, ISBN 3-89011-307-9. Data Becker GMBH, Merowingerstr. 30, 4000 Düsseldorf 1, Tel. 0211/310010.

Die Speicheraufteilung des C64

Erinnern Sie sich noch an die Aussage: der C64 verwalte mehr als 64 KByte? Wir beschlossen, die Klärung dieser Behauptung auf später zu verschieben. Jetzt ist es soweit.

Wegen der Struktur des Adressbusses kann unser C64 maximal 64 KByte auf einmal verwalten. Tatsache ist allerdings, daß unser Computer mehr Speicher besitzt. Es sind dies 64 KByte RAM (Schreib-/ Lesespeicher), 20 KByte ROM (Nur Lesespeicher) und 4 KByte, über die alle Zusatzchips (VIC, SID, CIAs) angesprochen werden. Klar, im ersten Augenblick ist es etwas unverständlich, wie diese zusätzlichen Speicherbereiche mit ins Gesamtkonzept passen. Des Rätsels Lösung: Gehen wir davon aus, daß es nicht nötig ist, immer den gesamten Schreib-/Lesespeicher zur Verfügung zu haben; denn wenn der Mikroprozessor ein Programm abarbeitet, interessiert ihn für die Ausführung in erster Linie nur der Bereich, in dem das Programm steht. Etwas anders sieht es mit den Speicherzellen aus, in die er Werte überträgt, oder aus denen er Daten holt. Sie können überall im Speicher verstreut liegen. Was spricht also dagegen, bei verschiedenen Datenbereichen zwischen einzelnen Bausteinen umzuschalten (Bank-switching)? Für Sie als Programmierer bedeutet dies allerdings: bevor Sie diese Speicherbereiche manipulieren, muß im Programm festgelegt sein, welches der Bauteile angesprochen werden soll. Damit dieses Verfahren für den Anwender nicht zu kompliziert wird, hat man sich darauf beschränkt, nur größere Blöcke (4 bis 8 KByte) in festgelegten Bereichen umschaltbar zu machen. Zur Verdeutlichung sind die erreichbaren Bausteine und deren Bereich in Abb.1 übereinander gezeichnet.



[1] **Das RAM des C64 mit den über Speicherstelle 1 erreichbaren ROMs**

Bei der Erklärung des Begriffs »Mikroprozessor« haben Sie schon den eingebauten Port kennengelernt. Er wird über Speicherstellen $00 und $01 angesprochen und steuert dieses Bank-switching; aber auch (damit es nicht zu einfach wird) die Kassettenoperationen. Betrachten wir uns zuerst $00, das Datenrichtungsregister. Es bestimmt, wie der Name auch sagt, die Datenrichtung der einzelnen Leitungen des Prozessor-Ports ($01):

Nach dem Einschalten wird diese Speicherstelle auf den Wert 47 gesetzt. Als Dualzahl entsteht damit »00101111«. Jedes der einzelnen Bits entspricht der Richtung des Prozessor-Ports; wobei ein gesetztes Bit (1) die entsprechende Leitung auf Ausgabe, ein gelöschtes Bit (0) auf Eingabe schaltet. Damit sind die drei niederwertigen Bits auf Ausgabe geschaltet (rechts). Und tatsächlich, über diese drei Leitungen wird ein spezielles Bauteil, zuständig fürs Bank-switching, gesteuert. Die drei nächsten Bits der Dualzahl sind für Kassettenoperationen zuständig. Die letzten Bits verdienen keinerlei Beachtung, da der Port nur sechs Leitungen besitzt.

Beachten Sie: Lassen Sie die Bits 0 bis 2 auf Ausgabe, da sie das Bank-switching steuern.

Allerdings nützt uns das Datenrichtungsregister allein nichts – denn wir müssen natürlich bestimmen können, welchen Zustand die Ausgangsleitungen und damit die Speicheraufteilung haben soll. Dafür ist Speicherstelle $01 zuständig. Ihr Wert nach dem Einschalten ist 55; d.h. dual »00110111«. Auch hier entspricht jedes Bit einer Port-Leitung. Im Unterschied zum Datenrichtungsregister schaltet der Inhalt des Datenregisters ($00) aber die Leitungen; bzw. teilt uns mit, ob die Leitungen High oder Low sind. Das Lesen der niedrigstwertigen Bits zeigt uns diesmal, wie der Speicher eingeteilt ist. Ein Beschreiben hat (endlich) eine Änderung der Speichereinteilung zu Folge:

Bit	Funktion
0	Basic-ROM =1, RAM = 0
1	Kernal-ROM =1, RAM = 0
2	I/O = 1, Zeichensatz = 0
3	Datenausgabe von Datasette
4	Taste von Datasette gedrückt = 0 nicht gedrückt = 1
5	Motor an = 1, Motor aus = 0
6	unbenutzt = 0
7	unbenutzt = 0

Die Kassettenfunktionen lassen wir, bis auf eine Bemerkung, dezent beiseite: Der Port-Ausgang Bit 5 ist über einen Transistor verstärkt, und kann (wenn kein Kassettenrecorder verwendet wird) direkt einen Motor oder ähnliches steuern.

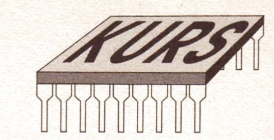
Wichtig zur Speichereinteilung: Ganz egal, wie Sie den C64 konfiguriert haben, ein Schreibzugriff wird niemals an einen ROM-Bereich geleitet, sondern grundsätzlich an das entsprechende RAM. Damit lassen sich einige Umschaltaktionen vermeiden. Beachten Sie bitte, daß Schreibzugriffe in das Zeichensatz-ROM den I/O-Bereich treffen, und damit seltsame Reaktionen hervorgerufen werden können.

Die wichtigsten Speicherkonfigurationen:

54 – Basic-ROM auf RAM geschaltet
53 – Basic-ROM und Kernal-ROM auf RAM geschaltet
52 – Basic-ROM, Kernal-ROM und Zeichen-ROM (bzw. I/O) auf RAM geschaltet
51 – blendet Zeichensatz ein

Zahlendarstellung in Assembler

Wir verwenden normalerweise bei Berechnungen das sog. Dezimalsystem. Schon in der Schule wurde uns beigebracht,



daß die Zahlen von 0 bis 9 jeweils eine Stelle bedeuten und die nächste Stelle jeweils das Zehnfache der Grundzahl ist. Mit diesem arabischen Zahlensystem können wir quasi im Schlafe umgehen (praktisch, da wir zehn Finger haben). Aber für unseren C64 ist genau dieses Zahlensystem das unpraktischste. Seine möglichen Zahlen sind, wie wir schon gehört haben, Null und Eins. In der Verknüpfung ist damit zwar »10« darstellbar (%000001010), aber »100« hat schon gar nichts mehr zu tun damit (%01100100), und gemein ist dabei, daß eine Speicherzelle 256 und nicht 100 Möglichkeiten hat. Das bedeutet für uns, daß wir erst umrechnen müssen, wenn wir wissen wollen, welchen Gesamtwert zwei aufeinanderfolgende Speicherstellen besitzen. Man nehme also den Inhalt der zweiten Speicherstelle, multipliziere ihn mit 256 und addiere den Inhalt der ersten dazu. Damit haben wir ihn - den Gesamtwert aus beiden Speicherstellen. Diese Methode funktioniert zwar, ist aber langwierig, fehlerträchtig und zudem noch kompliziert. Daher verwendet man zwei andere Systeme, die dem Konzept eines Computers näherkommen. Eines davon haben wir schon kennengelernt: das Binärsystem. Hier wird nur mit »0« und »1« argumentiert. Zur Unterscheidung von unserem Dezimalsystem kennzeichnet man diese Zahlen durch ein vorangestelltes %-Zeichen. Bei vielen Anwendungen ist dieses System durchaus sinnvoll. Denken wir an die Speichereinteilung. Hier entspricht eine 0-/1-Aussage jeweils einer Leitung des Prozessor-Ports. Aber spätestens, wenn wir wieder die zwei Zahlen aus den Speicherstellen verwenden wollen, wird diese Methode fürchterlich undurchsichtig. Es entsteht eine Zahl mit 16 Stellen, von denen jede 0 oder 1 sein kann. Das kann sich kein Mensch merken. Also verwendet man noch eine andere Methode - das Oktalsystem, auch Hexadezimalsystem genannt. Hier kann eine Stelle 16 Werte annehmen: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, und F. Die nächsthöhere hat daher auch die 16fache Wertigkeit. Damit läßt sich in zwei Stellen der Wert eines Bytes ausdrücken (16 x 16=256). Vier Stellen beinhalten also den gesamten Adressierungsbereich des C64 (256 x 256=56536). Hier wurden zwei Fliegen mit einer Klappe geschlagen:

1. Ganz egal welche Zelle des C-64-Speichers beschrieben werden soll, es genügen vier Stellen zur Darstellung (0000 bis FFFF)
2. Um den Gesamtwert aus mehreren Speicherstellen darzustellen, muß nicht mühsam berechnet werden, es genügt, die einzelnen Werte nebeneinanderzustellen.

Damit man dieses Zahlensystem von Dezimal- oder Binärzahlen unterscheiden kann, kennzeichnet man es durch ein vorangestelltes $-Zeichen.

Wie gebe ich Assemblerbefehle ein?

Nachdem Sie jetzt so viel von den Vorzügen des Assemblers, vom Mikroprozessor, von Speicher, Bank-switching und Zahlensystemen gehört haben, sind Sie bestimmt begierig, Ihr erstes Assemblerprogramm zu schreiben. Dazu benötigen Sie natürlich ein Werkzeug: den Assembler. Um komfortabel programmieren zu können befindet sich auf der beiliegenden Diskette der Hypra-Ass. Seine Funktionsbeschreibung finden Sie ab Seite 35. Mit Editor und Reassembler bietet er eine große Vielfalt an Verwendungsmöglichkeiten. Damit es aber am Anfang nicht zu kompliziert wird, bedienen wir uns des Assemblers im Monitor. Laden Sie daher den SMON von der beiliegenden Diskette mit

```
LOAD"SMON $C000",8,1
```

geben Sie anschließend NEW (<RETURN>) ein und starten Sie mit SYS49152 (<RETURN>). Der SMON begrüßt Sie mit der Anzeige der Register und wartet während der folgenden Erklärungen auf Ihre Eingaben.

Die ersten Befehle – LDA, STA, BRK und RTS

Das menschliche Gehirn hat dem Computer vieles voraus. Dazu gehört beispielsweise, daß ein Mensch allerlei Dinge gleichzeitig tun kann: gehen, sprechen, Musik hören, lächeln, Handbewegungen ausführen usw. Ihr C64 ist dazu nicht imstande. Er erledigt eine Aufgabe nach der anderen. Weil er das aber so schnell macht, hat es für uns den Anschein, es geschehe alles gleichzeitig. Das Maschinenprogramm ist die Kette solcher kleinen Aufgaben. Das erste Glied, das wir daraus kennenlernen wollen, ist der Befehl »LDA«.

Das bedeutet: Lade den Akkumulator. Alle Assembler-Befehlswörter bestehen aus drei Buchstaben (wie dieser hier auch). Es wurde schon erwähnt, daß einem solchen Befehl je eine 8-Bit-Codezahl entspricht. Das ist hier $A9 oder binär 10101001 oder schließlich dezimal 169. Die Codezahl muß in einem Speicherplatz stehen, z.B. in $1500 (entspricht dez. 5376). Assemblerlistings sehen dann folgendermaßen aus:

```
1500 LDA
```

Hier tritt also die Speicherplatznummer mit dem nachfolgenden Befehl anstelle der von Basic gewohnten Zeilennummer.

Es fehlt allerdings noch etwas Entscheidendes: Was soll denn in den Akku geladen werden? Genauso wie es in Basic Befehle gibt, die für sich alleine stehen können (CLR oder LIST), gibt es auch im Assembler solche Befehle. Weitaus häufiger sind allerdings andere, die ein »Argument« erfordern (in Basic z.B. PEEK (100); dabei ist 100 das Argument). In Assembler gibt es zwei Arten von Argumenten:

1. Argumente in 1-Byte-Format
2. Argumente in 2-Byte-Format

Bei einigen Befehlen existieren daher für ein einziges Befehlswort (hier LDA) 1-Byte-, 2-Byte- und 3-Byte-Befehle.

Das Argument von LDA ist also das, was in den Akku geladen werden soll. Laden wir daher eine »1« in den Akku. Dazu geben wir beim SMON ein:

```
A 1500
```

Diese Eingabe bringt SMON (nach <RETURN>) dazu, in den Assemblermodus zu gehen und hat zur Folge, daß »1500« am Bildschirm erscheint und der Cursor mit unwilligem Blinken zur weiteren Eingabe auffordert. Das tun Sie auch:

```
1500 LDA #01
```

nach <RETURN> wird der Befehl am Bildschirm gewandelt

```
1500 A9 01      LDA #01
```

und es erscheint »1502«. Dieser Wert sagt Ihnen die nächste zur Verfügung stehende Speicherstelle und erwartet wieder eine Eingabe:

```
1502 BRK
```

Sie verstehen nicht warum »BRK«? BREAK ist ein 1-Byte-Befehl (ohne Argument) und sagt dem Mikroprozessor, daß ein Programmabschnitt beendet ist und er zu einem festgelegten Programm verzweigen soll (in unserem Fall zum SMON). $1502 ist die nächste freie Speicherstelle, und wenn der Programmzähler nach dem LDA #01 auf 1502 deutet, erwartet der Mikroprozessor dort den nächsten Befehl. Wenn dort Unsinn steht, stürzt der Mikroprozessor im allgemeinen ab - je nachdem, welcher Code hier zufällig enthalten ist. Wir haben ja 256 Möglichkeiten ($00 bis $FF). Im Gegensatz zu Basic, wo man durch den Interpreter die Möglichkeit hat, Zeilennummern zu bauen, muß bei Maschine das Programm eine ununterbrochene Perlenschnur von Befehlen sein. Durch BRK läßt sich dieses Prinzip unterbrechen. Der Mikroprozessor unterbricht seine Arbeit und springt zurück zum Monitor.

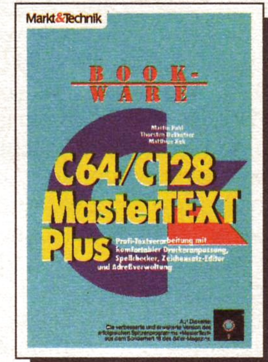
Mehr Fun

Withöft/Draheim

64'er – Großer Einsteiger-Kurs

Leichtverständlicher Einführungskurs in die Welt des C64. Die Autoren dokumentieren mit diesem Buch ihre lehrreichen Erfahrungen, Erfolge und auch Fehlschläge. So entstand ein Buch, das den Computer-Neuling Schritt für Schritt durch die Welt des C64 führt. Angefangen vom Auspacken und Anschließen des C64, über Basic-Programmierung bis zu PEEK- und POKE-Befehlen oder der Sprite-Programmierung wird das für den Einsteiger Wissenswerte über den C64 behandelt. Mit allen großen Beispielprogrammen auf der beiliegenden Diskette.
1988, 236 S., inkl. Diskette
ISBN 3-89090-668-0, DM 29,90

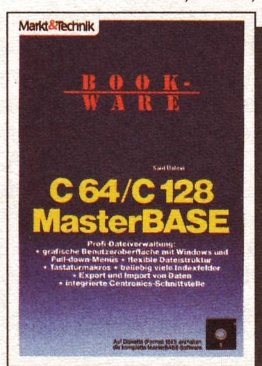
Über 50.000 verkaufte Exemplare!



Software zum Buchpreis: Profi-Textverarbeitung mit komfortabler Druckeranpassung, Spellchecker, Zeichensatz-Editor und Adreßverwaltung. Außerdem: individuelle Farbgebung, komprimiertes Dateiformat, freie Tastenbelegung, Silbentrennung und ... und ... und ...
1988, 201 S., inkl. Diskette
ISBN 3-89090-527-7, DM 59,–*



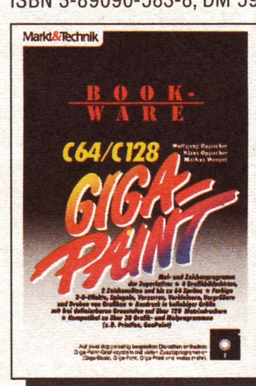
C64/C128-Software zum Buchpreis: eines der professionellsten 3D-CAD-Programme, das Grafiken von starker Räumlichkeit und Faszination erstellt. Ermöglicht ruckfreie Filme mit bis zu 24 Bildern pro Sekunde. Die Bildkonstruktion erfolgt mit dem Joystick.
1986, 183 S., inkl. Disketten
ISBN 3-89090-409-2, DM 49,–*



Software zum Buchpreis. Profi-Datenbank: grafische Benutzeroberfläche mit Windows und Pull-Down-Menüs, flexible Dateistruktur, Tastaturmakros, beliebig viele Indexfelder, Export und Import von Daten, integrierte Centronics-Schnittstelle.
1988, 155 S., inkl. Diskette
ISBN 3-89090-583-8, DM 59,–*



CAD-Programm für technisches Zeichnen, Malen in Farbe, Glückwunschkarten und sogar kleine Trickfilme. Mit 234 Seiten Dokumentation und vielen Anwendungsbeispielen. Außerdem ein komfortables Druckeranpassungsprogramm.
1985, 234 S., inkl. Diskette
ISBN 3-89090-136-0, DM 48,–



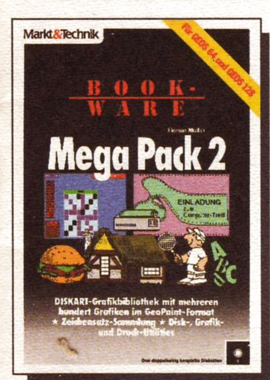
Super-Malprogramm zum Buchpreis: 4 Grafikbildschirme, 2 Zeichensätze und bis zu 64 Sprites; farbige 3D-Effekte; Grafiken spiegeln, verzerren und drehen; Ausdruck in beliebiger Größe mit frei definierbaren Graustufen. Zahlreiche Zusatzprogramme.
1988, 261 S., inkl. 2 Disketten
ISBN 3-89090-619-2, DM 59,–*



Software zum Buchpreis. Die nützliche Ergänzung für GEOS 64 und GEOS 128: Grafikbibliothek, Zeichensatzsammlung, Standardgrafiken importieren, Printfox-Zeichensätze einlesen und Druckeranpassungsprogramm.
1989, 160 S., inkl. 3 Disketten
ISBN 3-89090-772-5, DM 59,–*

für den C 64

für Fans!



Die Fortsetzung von Mega Pack 1: insgesamt 1 MByte auf 3 Disketten mit mehr als 500 DISKART-Grafiken im GeoPaint-Format, 33 GEOS-Zeichensätzen und prima Grafik-, Disk- und Druck-Utilities. Die Dateien sind mit allen deutschen GEOS-Versionen verwendbar.
1989, 177 S., inkl. 3 Disketten
ISBN 3-89090-350-9, DM 59,–*



Eine Kombination aus Assembler-Kurs im Buch und Software auf Diskette: komplettes Entwicklungspaket mit Makroassembler, Maschinensprachenmonitor, Reassembler, Einzelschrittsimulator. Die Entwicklung von eigenen Assemblerprogrammen wird so zu einer leichten Übung.
1988, 314 S., inkl. Diskette
ISBN 3-89090-571-4, DM 59,–



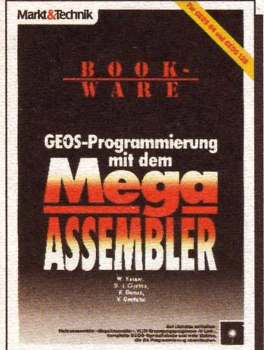
Eine Zusammenstellung aller Kniffe rund um den C64 in Basic und Maschinensprache. Leichtverständliche Erklärungen für Einsteiger und neueste Anregungen für Insider. Mit Beispielprogrammen und besten Utilities auf Diskette. Nach der Lektüre sind Sie der Profi auf dem C 64.
1988, 439 S., inkl. Diskette
ISBN 3-89090-499-8, DM 59,–

64'er Spielesammlung, Band 1: ★ Balliard ★ Maze ★ The Way ★ Schiffeversenken ★ Vager 3 ★ Firebug ★ Pirat ★ Handel ★ Wirtschaftsmanager ★ Börse ★ Vier gewinnt ★ Vier in Vier ★ Brainstorm ★ Schach ★ Zauberwürfel. Die Entführung in eine abenteuerliche Spielewelt.
1987, 115 S., inkl. Diskette
ISBN 3-89090-429-7, DM 39,–*

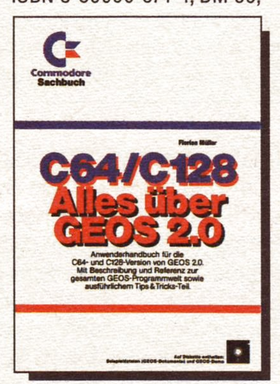


64'er Spielesammlung, Band 2: ★ Golf ★ Billard ★ Tonti ★ Zauberschloß ★ Freiheit ★ Steel Slab ★ Space Invader ★ Apocalypse Now ★ Block Out ★ Aquator ★ Tödliches Dioxin ★ Libra ★ Dasher ★ Bundesliga-Manager. Für alle, die vom Spielen nicht genug bekommen.
1987, 98 S., inkl. Diskette
ISBN 3-89090-428-9, DM 39,–*





Der ideale Einstieg in die Assembler-Programmierung unter GEOS. Damit kann jeder C64/C128-Benutzer eigene GEOS-Anwendungen, Desk-Accessories und VLIR-Programme schreiben. Mit allen Routinen und Systemvariablen zum Nachschlagen.
1990, 432 S., inkl. Diskette
ISBN 3-89090-247-2, DM 89,–*



Anwenderhandbuch für GEOS 2.0 (C64/C128). Mit Beschreibung und Referenz zur gesamten GEOS-Programmwelt sowie vielen Tips & Tricks. Mit GEOS-Demo auf Diskette.
1989, 432 S., inkl. Diskette
ISBN 3-89090-808-X, DM 59,–



Eine leichtverständliche Einführung in die digitale Schaltungstechnik. Mit Platinenlayouts und Selbstbauanleitungen für Sprachausgabebaustein, 128-Kbyte-EPROM-Karte u. a. Mit Treibersoftware auf Diskette.
1987, 294 S., inkl. Diskette
ISBN 3-89090-389-4, DM 49,–



64'er Spielesammlung, Band 3: ★ Arabian Treasurehunt ★ Block'n'Bubble ★ Robo's Revenge ★ Race of the Bones ★ Quadranoid ★ Future Race ★ Risiko ★ Copter Flight ★ Asteroids 64 ★ Verminator ★ Der kleine Hobbit ★ Odyssey.
1988, 103 S., inkl. Diskette
ISBN 3-89090-596-X, DM 39,–*

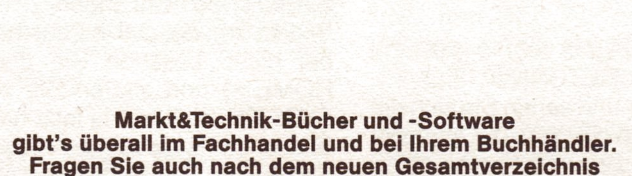


64'er Spielesammlung, Band 4: ★ Samurai: Kampfspiel mit Spitzengrafik ★ Omidar: mit einem Gorilla auf der Flucht ★ Bomb Runner: Minensuchkommando ★ Crillion: Mischung aus Boulder Dash und Arkanoid.
1988, 80 S., inkl. Diskette
ISBN 3-89090-703-2, DM 39,–*

*Unverbindliche Preisempfehlung



Spiele programmieren und dabei alles über Basic lernen. Die im Buch beschriebenen 21 Spiele sind als Listings abgedruckt und werden auf der beiliegenden Diskette gleich mitgeliefert.
1989, 209 S., inkl. Diskette
ISBN 3-89090-701-6, DM 39,–



Markt&Technik-Bücher und -Software gibt's überall im Fachhandel und bei Ihrem Buchhändler. Fragen Sie auch nach dem neuen Gesamtverzeichnis mit über 500 aktuellen Computer-Büchern und Software.





2006/108

Nachdem Sie die Eingabe wieder mit <RETURN> bestätigt haben, steht jetzt am Bildschirm:

```
1500 A9 01      LDA #01
1502 00         BRK
1503
```

Damit sind wir allerdings noch nicht fertig, denn SMON muß noch mitgeteilt werden, daß der Assemblierungsvorgang beendet ist. Geben Sie ein:

```
1503 F
```

danach wiederholt SMON die eingegebenen Befehle und unser kleines Programm ist assembliert.

Beachten Sie dabei, daß SMON alle Eingaben in hexadezimaler Schreibweise erwartet. Bei anderen Monitoren kann dies anders sein. Beim Hypra-Ass ist es sogar möglich, die unterschiedlichen Zahlenformate mit vorangestelltem Kennzeichen zu mischen ($ für hexadezimal, % für Binär). Doch was bedeutet # in unserem Assemblerbefehl? Es gibt viele Arten, den Akku zu laden. Direkt mit einer Zahl - wie hier - aber auch mit dem Inhalt einer Speicherstelle. Man spricht dabei von »Adressierung«. Es gibt eine ganze Menge davon, und jede wird auf eindeutige Art und Weise gekennzeichnet. Wenn wir mit unserem Akku direkt eine Zahl laden, dann ist das die »unmittelbare« Adressierung (immediate) und die kennzeichnet man mit #.

unmittelbare Adressierung (immediate)

Bei dieser Adressierungsart muß im Maschinencode unmittelbar nach dem Befehl der Wert erscheinen, der behandelt wird: Beispielsweise steht für »LDA #01« daher »$A9 $01«; also zuerst der Code für den Befehl, dann der zu ladende Wert.

Das kurze Listing ist übrigens mit auf Diskette und wird im SMON folgendermaßen geladen

```
L"LI.01"
```

danach läßt es sich mit

```
D 1500 1503
```

dissassemblieren. Starten wir einmal unser kleines Programm:

```
G 1500
```

Unmittelbar danach werden die Register angezeigt. Der Programmzähler (PC) steht auf 1503, im Akku (AC) steht 01, alle Flaggen außer der Breakflagge sind Null (die unbenutzte Flagge steht immer auf 1). Jetzt ändern wir das Argument in »LDA #00«. Geben Sie dazu ein:

```
D 1500 1503
```

Danach erscheint Ihr Listing:

```
1500 A9 01      LDA #01
1502 00         BRK
------------------
```

Ändern Sie in der Zeile 1500 neben LDA das #01 in #00 und bestätigen Sie diese Änderung mit <RETURN>.

Starten Sie nun wieder mit »G 1500« und sehen Sie sich die Register an: Programmzähler 1503, Akku jetzt 00, aber bei den Flaggen hat sich etwas geändert: Die Zero-Flagge ist auf 1 gesetzt. Wir sehen daran: die Zero-Flag ist so lange Null, bis in einer Operation (in unserem Fall der Akku) das Ergebnis oder ein Ladeprozeß Null ergibt. Ändern Sie jetzt das Programm in »LDA #FF«, oder laden Sie »LI.02«. Starten Sie danach wieder mit »G1500«. Natürlich steht FF im Akku, nur bei den Flaggen ist etwas merkwürdiges passiert: die Vorzeichenflagge steht auf 1. Das bedeutet, im Akku soll eine negative Zahl stehen! Wir wissen aber alle, daß $FF = dez. 255 ist. Es liegt allerdings kein Fehler vor: Immer wenn in einer Zahl das Bit 7 gleich 1 ist, geht zugleich die Vorzeichenflagge auf 1. Die Lösung des Rätsels sind die sog. negativen Binärzahlen. Bei Ihnen gilt eine Zahl als negativ, wenn Bit 7 gesetzt ist und als positiv, wenn Bit 7 gleich 0 ist. Sehen Sie sich dazu auch das Befehlsposter auf Seite 26/27 an.

Der LDA-Befehl beeinflußt die Vorzeichen- und die Zero-Flagge

Der nächste Befehl ist die Umkehrung von LDA und heißt »STA« (STore Accumulator), also lege den Akkumulatorinhalt ab. Wie Sie sich denken können, muß auch hier ein Argument auftauchen: nämlich, wohin der Akkuinhalt abgelegt wird. Wir legen den Akkuinhalt in die erste Bildschirmspalte ($0400). Damit müßte nach dem Programmstart ein Zeichen auf dem Bildschirm erscheinen. Laden Sie dazu das dritte Listing von Diskette

```
L"LI.03"
```

und sehen Sie es sich mit »D1500150B« an. Mit STA in Zeile1502 lernen wir eine neue Adressierungsart kennen: die »absolute« Adressierung. Man erkennt Sie daran, daß keine Zusätze verwendet werden (STA 0400). Die Adresse 0400, in die der Akku abgelegt wird, ist nicht in einem Byte darstellbar, sondern wird aufgeteilt in zwei Bytes. Im Speicher steht jetzt ab 1502:

```
8D 00 04
```

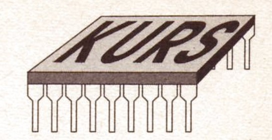
absolute Adressierung

Bei dieser Adressierungsart erscheint nach dem Code für den Befehl die Speicherposition, in der das zu behandelnde Argument steht. Die Darstellung erfolgt im Low-/High-Byte-Format. z.B. steht für »LDA 0400« der Code »$AD $00 $04« im Speicher.

»8D« ist der Befehlscode für STA, »00« ist das niederwertige Byte (LSB) und »04« das höherwertige Byte (MSB). Es liegt also ein 3-Byte-Befehl vor und der nächste Befehl darf ab $1505 beginnen. Von Basic her wissen wir, daß 1 der Bildschirmcode für »A« ist. Um dieses »A« aber vom Hintergrund abzuheben, bestimmen wir, daß es schwarz (Farbe 0) erscheinen soll (LDA #00), und schreiben diesen Wert an die entsprechende Position ($D800) ins Farbregister (STA D800). Die nächste freie Speicherposition ist jetzt $150A. Damit unser Programm abgeschlossen ist, steht an dieser Position »BRK«. Es könnte hier auch RTS (ReTurn from Subroutine), also Rückkehr aus dem Unterprogramm stehen. Auch damit ist das Programm abgeschlossen. Allerdings springt der Mikroprozessor nicht zurück zum SMON, sondern zu der Stelle, von der wir den SMON gestartet haben, nach Basic. Ändern Sie doch mal in 150A den Befehl »BRK« in »RTS« und starten Sie mit G1500. Wenn Sie mit dem Cursor nicht zu weit unten waren, sehen Sie jetzt (wie nicht anders zu erwarten, ein schwarzes »A« am Bildschirm, der evtl. erscheinende SYNTAX ERROR stört dabei nicht. Löschen Sie den Bildschirm (<SHIFT CLR/HOME>) und starten Sie erneut, diesmal aus Basic mit SYS5376. Spätestens jetzt haben Sie ihr schwarzes »A« am Bildschirm und das gewohnte READY. Gehen wir zurück zum SMON (mit SYS49152); und sehen wir uns nochmals den Befehl RTS an (mit D150A 150A). Wie deutlich sichtbar ist dies ein 1-Byte-Befehl ($60). Auch hier spricht man von einer Adressierungsart, nämlich von »implizit«. Man erkennt sie am Fehlen des Arguments. Die Adresse ist implizit, d.h. im Befehl selbst enthalten. Für den Prozessor bedeutet dies: er holt sich vom Stapel die oberste Adresse. Diese wurde dort abgelegt, als der Prozessor mit SYS aus Basic sprang.

Adressierungsart : implizit

Hier ist mit dem Befehlscode der komplette Befehl beschrieben. D.h. er besteht aus einem Byte, benötigt also kein Argument.



Vier weitere Befehle: LDX, STX, LDY, STY

Die Kombination von LDA mit STA ist vergleichbar mit dem POKE-Befehl aus Basic. Man kann allerdings in Assembler nicht direkt eine Speicherstelle beschreiben, sondern muß den Umweg über ein Register machen. Eines davon haben wir bereits kennengelernt - den Akku. Außer ihm eignen sich die beiden Hilfsregister x und y. Die Befehle für das x-Register lauten: LDX (LoaD X - lade x-Register) und STX (STore X - lege x-Register-Inhalt ab). Für das y-Register ist zuständig: LDY (LoaD Y - lade y-Register) und STY (STore Y - lege y-Register-Inhalt ab). Probieren Sie das doch mal an Listing 4 (L″LI.04″) aus. Es läßt sich mit »D1500 1519« disassemblieren, mit G1500 starten und bringt »ABA« auf den Bildschirm. Dabei ist das x-Register dreimal ausgelesen worden, der Akku zweimal und das y-Register einmal. Sie sehen, daß die Registerinhalte durch STA, STX und STY nicht verändert werden.

Ausführungszeiten der Befehle

Wenn Sie jetzt die Tabelle unten betrachten oder auf S. 26/27 aufblättern und sich die Ausführungszeiten (Zyklen) für die einzelnen Befehle ansehen (unmittelbar = imm, absolut = Abs), müßten Sie nachrechnen können, wie lange unser letztes Programm zur Ausführung gebraucht hat (1 Zyklus = ca. 1 Mikrosekunde). Es waren 48 Mikrosekunden (0,000048 Sekunden). Ein vergleichbares Basic-Programm benötigt für ein vergleichbares Programm 0,05 Sekunden, also etwa tausendmal so lange.

Befehls-wort	Adressie-rung	Byte-anzahl	Code Hex	Dez	Dauer in Taktzyklen	Beein-flussung von Flaggen
INX	implizit	1	E8	232	2	N,Z
INY	implizit	1	C8	200	2	N,Z
INC	absolut	3	EE	238	6	N,Z
DEX	implizit	1	CA	202	2	N,Z
DEY	implizit	1	88	136	2	N,Z
DEC	absolut	3	CE	206	6	N,Z
SED	implizit	1	F8	248	2	1 → D
CLD	implizit	1	D8	216	2	0 → D
BNE	relativ	2	D0	208	2 +1 bei Verzweigung +2 bei Überschreiten einer Seitengrenze	—

Kurzübersicht der Befehle fürs Zählen

Wir zählen: INX, INY, INC, DEX, DEY und DEC

Sie können »zählen« wörtlich nehmen, denn alle diese Befehle haben eines gemeinsam: sie erhöhen oder erniedrigen entweder Register oder Speicherstellen um »1«. Das wird schon beim Ausschreiben der Abkürzungen erkennbar: INX heißt INcrement x-Register, also »erhöhe das x-Register um 1«. Es wird sicherlich einleuchten, daß INY (INcrement y-Register) das gleiche mit dem y-Register macht. Ein wenig diffuser ist INC - INCrement memory (zähle zum Inhalt einer Speicherstelle 1 hinzu). INX und INY enthalten alles, was dem Computer zu sagen ist und sind daher 1-Byte-Befehle mit impliziter Adressierung. Mit INC verhält es sich anders. Hier muß dem Mikroprozessor noch mitgeteilt werden, welche Speicherstelle gemeint ist. Das läßt diesen Befehl zu einem 3-Byte-Befehl werden.

Die Umkehrung dieser Befehle lautet DEX (DEcrement x-Register), DEY (DEcrement y-Register) und DEC (DECrement memory). Da »decrement« »um eins verringern« bedeutet, erniedrigt der Mikroprozessor auch bei Anwendung dieser Befehle jeweils entweder x-Register, y-Register oder eine Speicherstelle um eins. Für die Adressierungsart und Anzahl der Bytes gilt das gleiche wie bei INX, INY und INC. Sehen wir uns dafür das Beispiel auf Diskette an (L″LI.05″ und D 15001519).

Wenn Sie das Programm mit G1500 starten (der Bildschirm darf nicht scrollen), erscheint in der linken oberen Ecke »ABA« in schwarzer Schrift. Was ist geschehen? Wir haben den Inhalt des Akku (0 = Farbcode für Schwarz) ins Farbregister geschrieben (ab $D800), dann den Inhalt des x-Registers (1 = Code für den Buchstaben »A«) in die erste Bildschirm-Speicherzelle. Anschließend wurde das x-Register um 1 erhöht (2 = Code für »B«) und dieser Inhalt in die zweite Bildschirmzelle geschrieben. Außerdem mußte dieser Bildschirm-Farbspeicherplatz mit dem Farbcode 0 belegt werden. Durch DEX wurde das x-Register um 1 reduziert, somit wieder ein »A« erzeugt und in die dritte Bildschirmstelle abgelegt.

Es ist Ihnen sicher aufgefallen, daß man auf diese Weise Abläufe mitzählen kann. Soll z.B. ein Vorgang 20mal wiederholt werden, schreibt man ins x-Register (möglich ist auch das y-Register oder eine Speicherstelle) den Anfangswert 0, läßt den Computer eine Arbeit ausführen, erhöht das Register oder die Speicherstelle (mit INX, INY oder INC). Anschließend prüft man, ob dieser Inhalt schon 20 geworden ist usw. Nun sollten wir uns aber grundsätzlich vor Augen halten: Ein Register oder eine Speicherstelle kann nur Werte von 0 bis 255 erhalten. Was passiert also, wenn wir weiterzählen? Für ein Beispiel geben Sie ein:

```
1500 LDX #FF
1502 INX
1503 BRK
1504 F
```

und starten mit G1500. Sie werden sehen, daß 255 + 1 Null ergibt (siehe XR). Allerdings ist die Zero-Flagge gesetzt. Ein Überlauf wird nicht angezeigt (obwohl einer stattfindet). Das gleiche passiert, wenn wir herabzählen:

```
1500 LDY  #01
1502  DEY
1504  F
```

Auch hier ist nach dem Ablauf der Routine die Zero-Flagge gesetzt. Es sei verraten, daß die Befehle INX, DEX, INY, DEY, INC und DEC nur zwei Flaggen beeinflussen: die Zero-Flagge und die Negativ-Flagge. Beachten Sie, alle anderen Flaggen bleiben unverändert.

Befehls-wort	Adressierung	Byte-anzahl	Code HEX	Code DEZ	Dauer in Takt-zyklen	Beein-flussung von Flaggen
LDA	unmittelbar	2	A9	169	2	N, Z
	absolut	3	AD	173	4	N, Z
LDX	unmittelbar	2	A2	162	2	N, Z
	absolut	3	AE	174	4	N, Z
LDY	unmittelbar	2	A0	160	2	N, Z
	absolut	3	AC	172	4	N, Z
STA	absolut	3	8D	141	4	keine
STX	absolut	3	8E	142	4	keine
STY	absolut	3	8C	140	4	keine
RTS	implizit	1	60	96	6	keine

Die Ausführungszeiten der ersten Befehle

BRANCH-Befehle

Wir haben inzwischen schon etliche Befehle kennengelernt. Wissen inzwischen auch, daß die meisten davon einige Flaggen beeinflussen. Na und, werden Sie sagen, denn was Sie Erstaunliches damit anfangen können, ist Ihnen noch nicht bekannt. Wenn Sie mit dem jetzigen Wissen von 255 auf 0 herabzählen möchten, bleibt nichts anderes übrig, als zuerst z.B. das x-Register mit $FF zu laden und danach 255 mal DEX zu schreiben. Mal abgesehen davon, daß eine Menge Speicherplatz verbraucht wird, sind Sie eine ganze Weile mit dem Eintippen beschäftigt. Sie haben es sich sicherlich schon gedacht: es gibt eine schnellere und bessere Methode. Um sie kennenzulernen, verwenden wir einige neue Befehle. Der erste davon ist BNE - Branch if Not Equal zero, oder verzweige, wenn ungleich Null. Sie ahnen es sicher: Dieser Befehl hat etwas mit der Zero-Flagge zu tun. Genauer gesagt, es wird zu einer angegebenen Adresse gesprungen, wenn die Zero-Flagge nicht gesetzt ist (= 0). Sehen wir uns dazu Listing 6 von der Diskette an (L"LI.06" und D1500150B). Zunächst werden das x- und das y-Register mit dem Ausgangswert $FF geladen (initialisiert). Mit DEY wird dann das y-Register um 1 herabgezählt (ergibt $FE). Die Zero-Flagge ist das der »0« (Klar, das Register enthält nicht »0«). Daher wird bei der nachfolgenden Überprüfung durch BNE in die danach festgelegte Speicherposition ($F504) verzweigt. Dort steht DEY, worauf das y-Register wieder um 1 erniedrigt wird. Dieses Spiel wiederholt sich nun so lange, bis endlich »0« im y-Register steht. Zugleich geht die Zero-Flagge auf »1«. Damit verzweigt der BNE-Befehl nicht mehr - der nächste Befehl (DEX) wird durchgeführt. Da allerdings jetzt das x-Register auf $FE steht, wird mit BNE nach 1502 verzweigt und das y-Register wieder mit $FF geladen. Die erste Schleife läuft wieder ab und ....

Wir haben hier zwei Schleifen ineinander verschachtelt. Die äußere davon wird 255mal durchlaufen, die innere 65025mal. Zur Verdeutlichung programmieren wir einmal diese Schleife in Basic:

```
100 FOR I = 255 TO 0 STEP-1
110 FOR J = 255 TO 0 STEP-1
120 NEXTJ
130 NEXTI
```

Diese Befehlsfolge bewirkt dasselbe wie unsere Assemblerroutine - eine Verzögerung im Programmablauf. Nur ist Basic ungleich langsamer. Starten Sie unsere Maschinenroutine mit G1500. Sie werden eine merkliche Verzögerung feststellen.

Noch längere Verzögerungen erhalten Sie, wenn Sie mehrere Schleifen ineinanderschachteln. Dabei verwenden Sie den DEC-Befehl.

Wozu Sie solche Verzögerungen brauchen, ist eigentlich klar: Wenn Sie z.B. einen Text vom Bildschirm lesen wollen, bevor das Programm weiterläuft, oder mit Peripherie arbeiten, die langsamer als der Computer ist, oder ... Allerdings sollte man erwähnen, daß es elegantere Methoden zur Verzögerung gibt, als ein Lahmlegen des Computers, doch dazu kommen wir etwas später.

BEQ ist die Umkehrung des BRANCH-Befehls BNE. Bei BEQ wird verzweigt, wenn die Zero-Flagge gleich »1« ist.

Anhand der Registeranzeige im SMON kennen Sie noch andere Flaggen. Die Carry- (C), die Negativ- (N) und die Überlauf-Flagge (V). Behandeln wir als nächstes die Carry-Flagge; für sie gibt es zwei Verzweigungs-(BRANCH-)Befehle:

1. BCC (Branch Carry Clear - verzweige, wenn Carry gelöscht) und

2. BCS (Branch Carry Set - verzweige, wenn Carry gesetzt).

Befehlswort	Adressierung	Byteanzahl	Code hex	Code dez	Dauer in Taktzyklen	Beeinflussung von Flaggen
ADC	unmittelbar	2	69	105	2	N,V,Z,C
	absolut	3	6D	109	4	
CLC	implizit	1	18	24	2	0 → C
SBC	unmittelbar	2	E9	233	2	N,V,Z,C
	absolut	3	ED	237	4	
SEC	implizit	1	38	56	2	1 → C
BEQ	relativ	2	F0		2	keine Änderung
BCC	relativ	2	90		2	keine Änderung
BCS	relativ	2	B0		2	keine Änderung
BMI	relativ	2	30		2	keine Änderung
BPL	relativ	2	10		2	keine Änderung
BVC	relativ	2	50		2	keine Änderung
BVS	relativ	2	70		2	keine Änderung

+1 bei Verzweigung
+2 bei Überschreiten einer Seitengrenze

Die Arithmetik auf einen Blick

Zusätzlich besteht die Möglichkeit, diese Flagge quasi von Hand zu setzen oder zu löschen:
1. SEC (SEt Carry - setze die Carry-Flagge)
2. CLC (CLear Carry - lösche die Carry-Flagge)

To carry heißt »tragen«. Hier stellt sich die Frage, was wird eigentlich getragen? Das zeigt sich am besten in einem Beispiel, in dem wir mit Binärzahlen 128 und 130 addieren:

```
    128      10000000
+   130      10000010
---------------------
    258  (1) 00000010
```

Das Ergebnis ist mit 258 zwar richtig, aber es paßt einfach nicht mehr in eine 8-Bit-Darstellung und damit auch nicht mehr in eine Speicherstelle. Ein Bit wurde überTRAGEN in ein extra dafür vorgesehenes Plätzchen - das Carry-Bit, auch Carry-Flagge genannt. Jedesmal, wenn so ein Übertrag bei einer Rechenoperation stattfindet, zeigt die Carry-Flagge eine »1«.

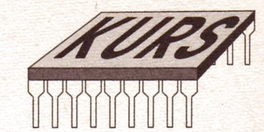
Je nach der Art Ihres Programmiervorhabens können Sie dieses Carry-Bit weiterverarbeiten. Es gibt auch Aufgaben, bei denen man es einfach vernachlässigen darf, oder solche, bei denen es in einer Rechnung weiterverwendet wird. Schließlich kann es uns noch anzeigen, wenn das größte Rechenergebnis %1111 1111 (255) sein darf.

Die Negativ-Flagge haben wir schon mal gestreift. Sie ist immer zugleich mit Bit 7 gesetzt und zeigt negative Zahlen an, wenn mit Zweierkomplementzahlen gearbeitet wird (pos. = 0 bis 127, neg. = 128 bis 255). Verzweigungsbefehle für diese Flagge sind:
1. BMI (Branch if MInus - springe, wenn Ergebnis minus, Negativ-Flagge = 1) und
2. BPL (Branch if PLus - springe, wenn Ergebnis Plus, Negativ-Flagge = 0).

Bleibt nur eine Flagge für BRANCH-Befehle übrig: die Overflow-Flag (V). Sie zeigt uns bei Addition zweier positiver Zahlen im Zweierkomplement ein falsches Ergebnis:

```
    64      01000000
+   66      01000010
--------------------
   -126     10000010
```

Das ist offensichtlich falsch. Bei der Addition ist durch Zusammenzählen der Bits 6 auch Bit 7 gesetzt worden. Da wir es aber mit der Zweierkomplement-Darstellung zu tun haben, wird die Überlauf-Flagge gesetzt. Leider ist die Sache nicht ganz so einfach, daß sie immer gesetzt wird, wenn von Bit 6 nach Bit 7 ein Übertrag stattfindet. Prinzipiell wird sie sich nur in folgenden zwei Fällen auf »1« ändern:



1. Es findet ein Übertrag von Bit 6 nach Bit 7 statt, aber kein äußerer Übertrag (Carry).
2. Es findet kein interner Übertrag von Bit 6 nach Bit 7 statt, aber ein äußerer Übertrag.

Merken kann man sich das am besten folgendermaßen: Immer dann, wenn quasi aus Versehen das Vorzeichenbit 7 verändert wurde, wird die V-Flagge auf 1 gesetzt. Das erfordert natürlich, daß man sich bei allen Operationen vorher überlegen muß, welche Fehler durch versehentliches Vorzeichenändern passieren können. Die Verzweigungsbefehle für die Overflow-Flagge sind:
1. BVS (Branch if oVerflow Set - springe, wenn V gesetzt) und
2. BVC (Branch if oVerflow Clear - springe, wenn V nicht gesetzt)

Arithmetik in Assembler – ADC

Der erste arithmetische Befehl, den wir kennenlernen, ist ADC (ADd with Carry - addiere mit Carry). Dazu addieren wir zuerst zwei Zahlen, die so klein sind, daß kein Überlauf stattfindet. Laden und betrachten Sie dazu Listing 7 (L"LI.07" und D1200 1209). Der Beginn der Befehlsfolge ist CLC, also lösche die Carry-Flagge. Warum? Nun, wir wissen nicht wie sie momentan aussieht und es gibt eine Menge Vorgänge in einem Assemblerprogramm, die diese Flagge beeinflussen. Weil jedoch ADC auch das Carry-Bit mitaddiert, sollte man dafür sorgen, daß es vor jeder Addition gelöscht ist. Dazu haben wir schon weiter oben den Befehl CLC kennengelernt. In unserem kleinen Programm wird der Akku mit $0C (12) geladen und mit ADC $07 addiert. Das Ergebnis wird in Speicherstelle $1244 abgelegt. Starten Sie doch mal das kleine Programm (G 1200), obwohl Sie diese Rechnung sicherlich schneller im Kopf berechnen könnten. »M12441245« zeigt das Ergebnis. Wie nicht anders zu erwarten, steht $13 (19) in Speicherstelle $1244. Es ist bei diesem Programm ziemlich mühsam, andere Werte einzusetzen, da die Werte mit unmittelbarer (immediate) Adressierung geladen und addiert werden. Beide Befehle können aber auch absolut adressiert werden. Dazu laden Sie Listing 8 (L"LI.08") und betrachten sich die Befehle (D1200120B). Die Behandlung wird bedeutend einfacher, wenn Sie sich mit M12401244 die Speicherstellen ansehen. Momentan stehen hier noch willkürliche Werte, aber Sie können durch Überschreiben von $1240 und $1242 zwei Zahlen vorgeben, die dann (nach G1200 und M12401244) in $1244 das Ergebnis zeigen. Was wir bis jetzt gemacht haben, ist die Addition zweier 8-Bit-Zahlen. Weitaus häufiger werden in der Praxis 16-Bit-Zahlen addiert. Laden und betrachten Sie dazu Listing 9 (L"LI.09" und D12001214). Für dieses Beispiel sind ab $1240 schon die Zahlen vorgegeben. Wie wir schon beim Programm-Counter gehört haben, teilen wir dazu die 16-Bit-Zahl in zwei 8-Bit-Zahlen (LSB und MSB). Bei uns stehen diese Zahlen schon in den Speicherstellen $1240 bis $1243. Es sind $0880 (2176) und $03F1 (1009). Fällt Ihnen auf, daß jeweils das niederwertige Byte vor dem höherwertigen steht? $0880 steht als »80 08« und $03F1 als »F1 03« im Speicher. Obwohl wir es natürlich auch anders programmieren könnten, ist diese Schreibweise äußerst sinnvoll, da bestimmte Sprungbefehle, von denen wir noch hören werden, dieses Format benötigen. Ein bißchen ausgewählt sind unsere Zahlen allerdings. Es wurde darauf geachtet, daß im höherwertigen Byte kein Überlauf vorkommen kann. Wenn Sie mit G1200 starten, werden zuerst die niederwertigen Bytes addiert: $80+$F1=$71 und unser Carry ist gesetzt. Danach kommen die höherwertigen Bytes an die Reihe: $08+$03+Carry=$0C. Damit steht das Ergebnis $0C71 (3185) ab der Speicherstelle $1244. Natürlich im Format »71 0C«. Falls Sie dem Programm nicht trauen, rechnen Sie doch einfach nach.

SBC – Subtrahieren

Sie werden es nicht glauben, subtrahieren oder addieren ist für den Mikroprozessor Jacke wie Weste; er hat nur einen Arbeitsschritt mehr zu erledigen:

Nehmen wir an, wir subtrahieren von der Zahl 100 das Argument 97. Das Ergebnis kennen Sie: »3«. Aber mit einem Trick kann man diese Rechnung auch durch Addition ausdrücken:

Nehmen wir zuerst das Argument (97 = %01100001). Danach bilden wir das Komplement davon. Das heißt aus jeder »0« wird eine »1« und umgekehrt. Das Ergebnis ist %10011110. Dazu addieren wir 100 (%01100100). Das Ergebnis ist jetzt 2 (%00000010), aber gemeinerweise nicht 3, wie es sein sollte. Wir müssen also 1 zusätzlich addieren. Zur Verdeutlichung die Zahlenfolge untereinander:

```
(0)10100001 = $61 Argument
------------------
(0)10011110 = $9E Komplement des Arguments
(0)01100100 = $64 + Zahl 100
------------------
(1)00000010 = $02 Ergebnis + Übertrag 9. Stelle
(0)00000001 = $03 Offset (1) dazu
==================
(0)00000011 richtiges Ergebnis
```

Eine genaue Erklärung, warum die »1« zum Ergebnis addiert werden muß, würde zu weit führen. Nehmen Sie es als gegeben.

Befehls-wort	Adressierung	Byte-zahl	Code Hex	Code Dez	Takt-zyklen	Beeinflussung von Flaggen
LDA	absolut,X	3	BD	189	4	N,Z
	0-page-abs,X	2	B5	181	4	N,Z
	absolut,Y	3	B9	185	4	N,Z
LDX	absolut,Y	3	BE	190	4	N,Z
	0-page-abs,Y	2	B6	182	4	N,Z
LDY	absolut,X	3	BC	188	4	N,Z
	0-page-abs,X	2	B4	180	4	N,Z
STA	absolut,X	3	9D	157	5	/
	absolut,Y	3	99	153	5	/
	0-page-abs,X	2	95	149	4	/
STX	0-page-abs,Y	2	96	150	4	/
STY	0-page-abs,X	2	94	148	4	/
INC	absolut,X	3	FE	254	7	N,Z
	0-page-abs,X	2	F6	246	6	N,Z
DEC	absolut,X	3	DE	222	7	N,Z
	0-page-abs,X	2	D6	214	6	N,Z
ADC	absolut,X	3	7D	125	4	N,V,Z,C
	absolut,Y	3	79	121	4	N,V,Z,C
	0-page-abs,X	2	75	117	4	N,V,Z,C
SBC	absolut,X	3	FD	253	4	N,V,Z,C
	absolut,Y	3	F9	249	4	N,V,Z,C
	0-page-abs,X	2	F5	245	4	N,V,Z,C
CMP	absolut,X	3	DD	221	4	N,Z,C
	absolut,Y	3	D9	217	4	N,Z,C
	0-page-abs,X	2	D5	213	4	N,Z,C
BIT	absolut	3	2C	44	4	N,V,Z
	0-page-abs.	2	24	36	3	N,V,Z
CLV	implizit	1	B8	184	2	V
NOP	implizit	1	EA	234	2	/
TAX	implizit	1	AA	170	2	N,Z
TAY	implizit	1	A8	168	2	N,Z
TXA	implizit	1	8A	138	2	N,Z
TYA	implizit	1	98	152	2	N,Z
JMP	absolut	3	4C	76	3	/
	indirekt	3	6C	108	5	/
JSR	absolut	3	20	32	6	/

Neue Adressierungsarten: Zeropage

Jetzt wird auch klar, warum wir dieses kleine Rechenbeispiel durchgearbeitet haben: Erinnern Sie sich noch an CLC und SEC? Wenn mit SEC die Carry-Flagge gesetzt ist, addiert der Mikroprozessor zusätzlich noch die »1«. Und da der

Mikroprozessor die Subtraktion intern genauso ausführt wie oben besprochen, müssen wir davor das Carry mit SEC setzen. Kommt ein Übertrag bei der Rechnung vor (z.B. 29-60= -31), wird das Carry gelöscht. Diese Befehlsfolge läßt sich auch in Assembler simulieren:

```
LDA     #  64 ;
Lädt Argument
EOR     #  FF ;
 Invertiert die Bits
SEC        ;
setzt Carry-Flagge
ADC     #  61 ;
addiert $64 + 1
BRK        ;
führt zurück in den Monitor
```

Stören Sie sich nicht an EOR, dieser Befehl macht eine Exclusiv-ODER-Verknüpfung mit $FF (%11111111). Damit werden alle Bits invertiert. Zur Erklärung kommen wir später. Interessanter ist, daß wir zuerst unsere zu subtrahierende Zahl in den Akku laden müssen. Da dies eine ungewöhnliche Rechenfolge ist, gibt es einen eigenen Befehl - SBC (SuBtract with Carry - subtrahiere mit der Carry-Flagge).

Unsere Rechnung lautet damit:

```
SEC
LDA #$64
SBC #$61
```

Nach der Ausführung dieser Befehlsfolge steht $03 im Akku und Carry bleibt gesetzt. Zum ausführlichen Test laden Sie Listing 10 von Diskette (L"Ll.10"). Betrachten Sie es mit »D1200120C«. Wenn Sie die Speicherstellen $1240 mit der Zahl und $1242 mit dem Subtrahenden überschreiben, erscheint das Ergebnis nach G1200 in $1244. Sie werden feststellen, daß SBC die Negativ-, Overflow-, Zero- und natürlich die Carry-Flagge beeinflußt.

Wenn Sie die Erklärung von SBC aufmerksam verfolgt haben, erklärt es sich von selbst, daß bei einer gelöschten Carry-Flagge das Ergebnis minus eins im Akku steht. Diese Eigenschaft nützt uns bei der 16-Bit-Subtraktion einiges. Laden und betrachten Sie dazu Listing 11 (L"Ll.11" und D12001215). In Speicherstelle $1240 gehört das Low-Byte in $1241 das High-Byte der 16-Bit-Zahl, von der die 16-Bit-Zahl (Low-Byte in $1243, High-Byte in $1244) subtrahiert werden soll. Das Ergebnis steht nach G1200 in $1244 (LSB) und $1245 (MSB). Bei dieser Rechnung wird ähnlich der Addition zuerst das Low-Byte behandelt, danach das High-Byte.

Vergleichen – CMP, CPX, CPY

Eigentlich sind diese Befehle nichts anderes, als die oben beschriebene Subtraktion - mit einer Ausnahme, das Rechenergebnis wird nicht festgehalten, es werden lediglich die Flaggen beeinflußt. Und für uns sehr wichtig, diese Vergleichsfunktionen lassen sich auch beim x- und y-Register verwenden. CMP (CoMPare to Accumulator - Vergleiche mit dem Akku-Inhalt) ist die entsprechende Funktion für den Akku und beeinflußt die Negativ-, Zero- und Carry-Flagge. Zusammen mit den Branch-Befehlen, lassen sich mit kurzen Programmen die kompliziertesten Abfragen aufbauen.

Zur Verdeutlichung noch einmal die Funktion der Branch-Befehle. Listing 12 wiederholt die Funktion von BEQ (L"Ll.12" und D2000200B). Hier wird ein Wert aus Speicherstelle $200B in den Akku geladen und wenn er Null ist, zur Position $200A verzweigt (BRK). Ist der Inhalt ungleich Null, wird $00 in den Akku geladen und wieder in $200B geschrieben. Diese Routine ergibt zunächst keinen Sinn. Darum schreiben wir sie um (A 2000):

```
2000 LDA C6
2002 BEQ 2000
2004 LDA #00
2006 STA C6
2008 BRK
2009 F
```

Danach starten wir mit G2000. Und siehe da, der Computer hängt. Doch das ist nur vermeintlich der Fall. Drücken Sie doch mal irgendeine Taste (außer SHIFT): Sie erhalten wieder die Registeranzeige des SMON.

Was ist passiert? Die Speicherzelle $C6 enthält die Information, ob eine Taste der Tastatur gedrückt wurde. Sie erinnern sich: Beim Ablauf von Basic-Programmen merkt sich der C 64 bis zu zehn Tastentips. Dazu wird jede 60stel Sekunde die Tastatur abgefragt (im Interrupt). Die Information, wie viele Tasten gedrückt wurden, steht in $C6 (198). Kein Tastendruck = 0, daher überprüfen wir mit BEQ diese Speicherstelle. Ist das Ereignis Tastendruck eingetreten ($C6 =1 oder größer), muß allerdings mitgeteilt werden, daß wir dies erkannt haben. Daher setzen wir $C6 wieder auf Null. Eine Tastatur-Warteschleife ist sinnvoller als eine Warteschleife rein auf Zeitbasis, da der User selbst entscheiden kann, wann es im Ablauf weitergeht. Doch es ist noch etwas Bemerkenswertes passiert. Sehen Sie sich die Zeile 2000 an. Hier steht jetzt:

```
2000 A5 C6    LDA C6
```

Unser Akku hat aus der Speicherstelle $C6 den Wert geladen. Nur ist der Befehl »LDA C6« kein 3-Byte-Befehl wie »LDA 3000«, sondern besteht aus den Bytes »A5 C6«. »C6« ist, wie unschwer zu erkennen ist, die Speicherstelle, aus der geladen werden soll, »A5« ist der Befehlscode. Diese Adressierungsart nennt man Zero-Page (oder in unserem Poster S.26/27 »0Page«). Sie funktioniert nur auf den ersten 256 Bytes (0 bis 255) und, Sie haben es richtig erkannt, hat ihren Namen von der Bezeichnung dieses Bereichs - Zero-Page.

Adressierungsart Zero - Page

Bezieht sich auf die ersten 256 Byte (Zero-Page) des Speicherbereichs. Da das High-Byte wegfällt entsteht ein 2-Byte-Befehl. »LDA C6« wird zu »$A5 $C6«.

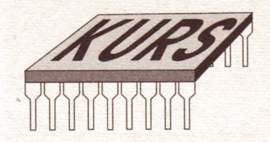
Zurück zu unserem Compare-Befehl:

Das Betriebssystem speichert natürlich nicht nur die Anzahl der Tastenimpulse, sondern auch welche Tasten gedrückt wurden (im ASCII-Code). Der dafür zuständige Bereich (Tastaturpuffer) reicht von dezimal 631 bis 640 ($0277 bis $0280). Also mal angenommen, wir laden unmittelbar nachdem ein Tastendruck aufgetreten ist, den Wert aus Speicherstelle 631 den ASCII-Code und vergleichen ihn mit dem Wert einer von uns gewünschten Taste, dann lassen sich im Programm schon einige Entscheidungen treffen:

```
A 2008
2008 LDA 0277
200B SEC
200C CMP #0D
200E BNE 2000
2010 BRK
2011 F
```

In Zeile 2008 laden wir das Hauptregister mit dem Wert aus der ersten Stelle des Tastaturpuffers. Danach setzen wir erstmal die Carry-Flagge. Anschließend vergleichen wir mit dem Wert $0D. Das ist dezimal 13, der Code für die RETURN-Taste. Hat unser Akku einen anderen Wert, verzweigen wir zurück zur Abfrage in Zeile 200 (ist eine Taste gedrückt?). Ansonsten führt der Programmablauf weiter. Starten Sie mal mit G2000. Nur <RETURN> bringt Sie zurück in den Monitor. Die ASCII-Codes finden Sie übrigens im Handbuch Ihres C64.

Dieselbe Wirkung läßt sich auch mit den beiden anderen Compare-Befehlen erreichen:



1. CPX (ComPare to register X – vergleiche mit x-Register) und

2. CPY (ComPare to register Y – vergleiche mit y-Register)
Auch hier muß vor dem Vergleich die Carry-Flagge gesetzt sein (SEC).

Relative Adressierung

Mit den Verzweigungsbefehlen BNE, BEQ, BPL, BMI, BVS, BCC und BCS haben wir sie zwar schon kennengelernt, aber noch nicht erläutert. Laden und betrachten Sie daher nochmal Listing 12 von Diskette (L″LI.12″ und D20002000B). Fällt Ihnen in Zeile 2003 auf, daß zwar »BEQ 200A« steht, im Code links daneben aber »F0 05«. Wenn wir »F0« als Befehlscode annehmen (was er auch ist), dann ist »05« aber zunächst nicht mit »200A« in Verbindung zu bringen; normal müßte doch »0A 20« neben dem Befehlscode stehen. Daß dem nicht so ist, liegt an der Benutzerfreundlichkeit des SMON. Er rechnet die absolute Adresse $200A in die relative ($05) um. Wir haben nur 2 Byte zur Verfügung. Eines für den Befehlscode und ein zweites für die relative Adressierung. Wenn wir dieses zweite Byte als Wert für die Abweichung zur momentanen Position des Programm-Counters (Offset) bezeichnen, wird seine Funktion schon deutlicher. Anhand dieses einen Bytes sehen wir auch, daß die Verzweigungen nicht allzuweit zur momentanen Position geschehen können – normalerweise 256 Speicherpositionen. Aber das stimmt nicht ganz. Auch hier ist klar, warum nicht: Ein Verzweigungsbefehl könnte sonst nur in eine Richtung erfolgen. Daß dem nicht so ist, ersehen wir aus Listing 13 (L″LI.13″ und D10001006). Hier haben wir den BNE-Befehl in Zeile 1003; er verzweigt laut SMON auf die Speicherposition $1002. Beim Befehlscode steht »D0 FD«, D0 für den Befehlscode und »FD« für den Offset.

Ist Ihnen etwas aufgefallen? Ein relativer Sprung nach vorne im Speicher wird mit einer positiven 1-Byte-Zahl markiert, ein Sprung nach hinten mit einer negativen. Zur Erinnerung: 1-Byte-Zahlen sind negativ, wenn Bit 7 gesetzt, also die Zahl größer als 127 ist. Ist Bit 7 gelöscht, wird die Zahl positiv angenommen. Jetzt verstehen Sie auch den Sinn von Zeile 2003: Springe $05 Positionen nach vorne im Speicher, wenn das Ereignis (BEQ = Zero-Flagge gesetzt) eintritt. Da der Programm-Counter immer auf den Beginn des nächsten Befehls deutet, springe nach $2005+$05=$200A. Um die relative Sprungadresse für Zeile 1003 zu berechnen, müssen wir zuerst die negative Zahl berechnen: $00-$FD=$03, wir müssen quasi über Eck rechnen. Damit ergibt sich für diese Operation ein Sprung nach $1005-$03=$1002. Sowohl der SMON als auch der Hypra-Ass nimmt Ihnen die Umrechnung ab.

Achtung: Bei der relativen Adressierung kann der Mikroprozessor maximal 127 Schritte nach vorne verzweigen. Nach rückwärts sind es 126 Schritte; 128 minus Befehlslänge des Branch-Befehls (=2).

Indizierte Adressierung

Indizieren heißt, etwas mit einem Index (Zeichen oder Nummer) zu versehen. Sie haben bestimmt schon mal ein Jahres-Inhaltsverzeichnis (z.B von unseren Stammheften) gesehen. Damit ist Ihnen auch schon eine Art der Indizierung in die Finger geraten. Wenn Sie einen Artikel gesucht und gefunden haben, steht daneben die Ausgabe (z.B. 1/91) und die Seitenzahl (z.B. S.32). Mit anderen Worten: Ihr gesuchter Artikel ist über die Ausgabe 1/91 mit der Seitenzahl 32 indiziert. Anhand dieser Indizierung nehmen Sie das Heft 1/91 in die Hand und schlagen Seite 32 auf. Dort befindet sich der gesuchte Artikel; und zwar dieser und kein anderer. So ähnlich können wir uns auch die Funktion der indizierten Adressierung vorstellen. Nehmen wir als Beispiel: LDA $1500,X. Man

Fortsetzung S. 21

So finden Sie die Programme auf der Diskette

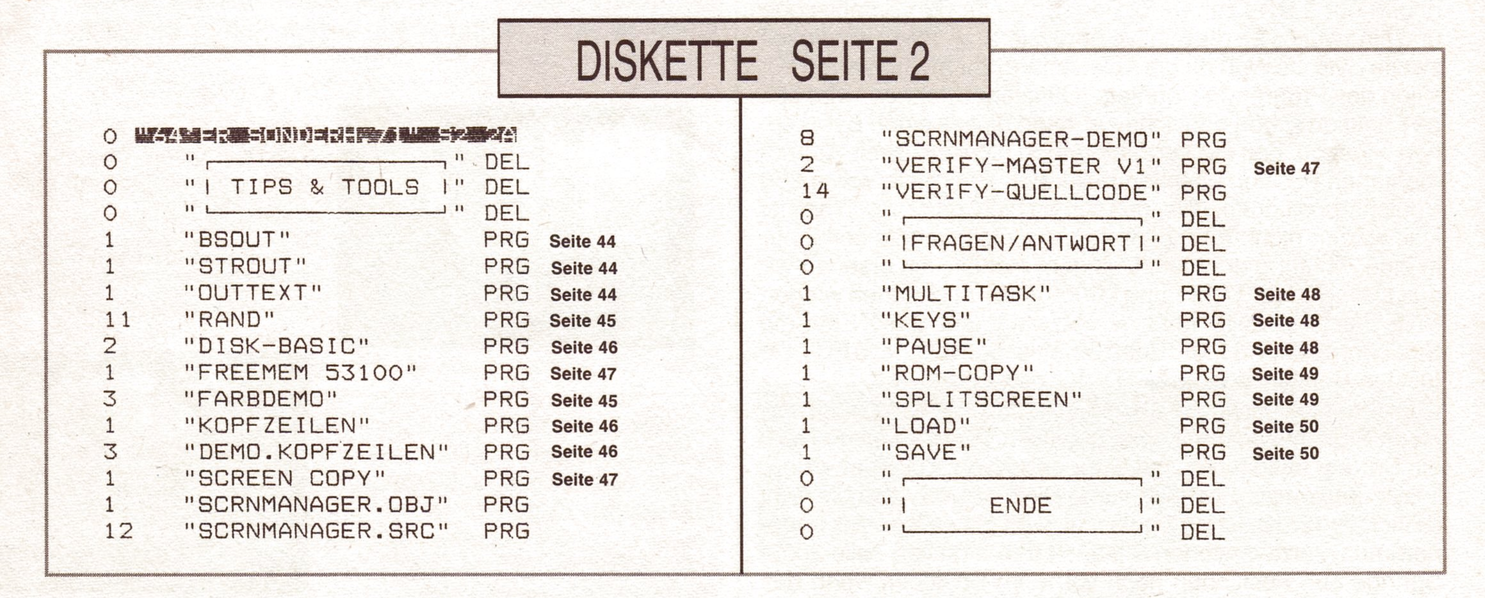
DISKETTE SEITE 1

```
0   "64'ER SONDERH.71" S1 2A
0      "┌──────────────┐" DEL
0      "|   MONITOR    |" DEL
0      "└──────────────┘" DEL
17     "SMON $C000"        PRG   Seite 30
17     "SMON $C000 II"     PRG
17     "SMON 128D"         PRG
12     "SMON+"             PRG
2      "NDISASS"           PRG   Seite 34
3      "FLOPPYMON"         PRG
1      "ILLDEMO"           PRG
0      "┌──────────────┐" DEL
0      "|  ASSEMBLER   |" DEL
0      "└──────────────┘" DEL
25     "HYPRA-ASS"         PRG   Seite 35
40     "HYPRA-ASS.EDI"     PRG
0      "┌──────────────┐" DEL
0      "| REASSEMBLER  |" DEL
0      "└──────────────┘" DEL
12     "REASS"             PRG   Seite 41
```

```
0      "┌──────────────┐" DEL
0      "|     KURS     |" DEL
0      "└──────────────┘" DEL
1      "LI.01"             PRG   Seite 4
1      "LI.02"             PRG
1      "LI.03"             PRG
1      "LI.04"             PRG
1      "LI.05"             PRG
1      "LI.06"             PRG
1      "LI.07"             PRG
1      "LI.08"             PRG
1      "LI.09"             PRG
1      "LI.10"             PRG
1      "LI.11"             PRG
1      "LI.12"             PRG
1      "LI.13"             PRG
1      "LI.14"             PRG
```

```
0      "┌──────────────┐" DEL
0      "|SONDERHEFT 35 |" DEL
0      "└──────────────┘" DEL
5      "DEMO VERSCHIEB."   PRG
1      "VERSCHIEB. 1."     PRG
1      "VERSCHIEB. 2."     PRG
4      "MODULSIM."         PRG
1      "RASTERINTER."      PRG
1      "PRG.TIMER-TEST"    PRG
3      "TIMER.SRC"         PRG
2      "ALARMUHR.OB"       PRG
41     "ALARMUHR.QUE"      PRG
0      "┌──────────────┐" DEL
0      "|   DISKETTE   |" DEL
0      "|  BEIDSEITIG  |" DEL
0      "|   BESPIELT   |" DEL
0      "└──────────────┘" DEL
```

DISKETTE SEITE 2



```
0   "64'ER SONDERH.71" S2 2A
0      "┌──────────────┐" DEL
0      "| TIPS & TOOLS |" DEL
0      "└──────────────┘" DEL
1      "BSOUT"             PRG   Seite 44
1      "STROUT"            PRG   Seite 44
1      "OUTTEXT"           PRG   Seite 44
11     "RAND"              PRG   Seite 45
2      "DISK-BASIC"        PRG   Seite 46
1      "FREEMEM 53100"     PRG   Seite 47
3      "FARBDEMO"          PRG   Seite 45
1      "KOPFZEILEN"        PRG   Seite 46
3      "DEMO.KOPFZEILEN"   PRG   Seite 46
1      "SCREEN COPY"       PRG   Seite 47
1      "SCRNMANAGER.OBJ"   PRG
12     "SCRNMANAGER.SRC"   PRG
```

```
8      "SCRNMANAGER-DEMO"  PRG
2      "VERIFY-MASTER V1"  PRG   Seite 47
14     "VERIFY-QUELLCODE"  PRG
0      "┌──────────────┐" DEL
0      "|FRAGEN/ANTWORT|" DEL
0      "└──────────────┘" DEL
1      "MULTITASK"         PRG   Seite 48
1      "KEYS"              PRG   Seite 48
1      "PAUSE"             PRG   Seite 48
1      "ROM-COPY"          PRG   Seite 49
1      "SPLITSCREEN"       PRG   Seite 49
1      "LOAD"              PRG   Seite 50
1      "SAVE"              PRG   Seite 50
0      "┌──────────────┐" DEL
0      "|     ENDE     |" DEL
0      "└──────────────┘" DEL
```

WICHTIGE HINWEISE

zur beiliegenden Diskette:

Aus den Erfahrungen der bisherigen Sonderhefte mit Diskette wollen wir ein paar Tips an Sie weitergeben:

1 Bevor Sie mit den Programmen auf der Diskette arbeiten, sollten Sie unbedingt eine Sicherheitskopie der Diskette anlegen. Verwenden Sie dazu ein beliebiges Kopierprogramm, das eine komplette Diskettenseite dupliziert.

2 Auf der Originaldiskette ist wegen der umfangreichen Programme nur wenig Speicherplatz frei. Dies führt bei den Anwendungen, die Daten auf die Diskette speichern, zu Speicherplatz-Problemen. Kopieren Sie daher das Programm, mit dem Sie arbeiten wollen, mit einem File-Copy-Programm auf eine leere, formatierte Diskette und nutzen Sie diese als Arbeitsdiskette.

3 Die Rückseite der Originaldiskette ist schreibgeschützt. Wenn Sie auf dieser Seite speichern wollen, müssen Sie vorher mit einem Diskettenlocher eine Kerbe an der linken oberen Seite der Diskette anbringen, um den Schreibschutz zu entfernen. Probleme lassen sich von vornherein vermeiden, wenn Sie die Hinweise unter Punkt 2 beachten.

ALLE PROGRAMME
aus diesem Heft



HIER

Assembler

Herausgeber: Carl-Franz von Quadt, Otmar Weber

Redaktionsdirektor: Dr. Manfred Gindle

Chefredakteur: Georg Klinge – verantwortlich für den redaktionellen Teil
Stellv. Chefredakteur: Arnd Wängler
Textchef: Jens Maasberg
Produktion: Andrea Pfliegensdörfer
Redaktion: Harald Beiler (bl), Herbert Großer (gr)
Redaktionsassistenz: Sylvia Wilhelm, Birgit Misera (089/4613202)
Telefax: 089/4613-5001
Alle Artikel sind mit dem Kurzzeichen des Redakteurs und/oder mit dem Namen des Autors/Mitarbeiters gekennzeichnet

Manuskripteinsendungen: Manuskripte und Programmlistings werden gerne von der Redaktion angenommen. Sie müssen frei sein von Rechten Dritter. Sollten sie auch an anderer Stelle zur Veröffentlichung oder gewerblichen Nutzung angeboten worden sein, muß dies angegeben werden. Mit der Einsendung von Manuskripten und Listings gibt der Verfasser die Zustimmung zum Abdruck in von der Markt&Technik Verlag AG herausgegebenen Publikationen und zur Vervielfältigung der Programmlistings auf Datenträger. Mit der Einsendung von Bauanleitungen gibt der Einsender die Zustimmung zum Abdruck in von Markt&Technik Verlag AG verlegten Publikationen. Honorare nach Vereinbarung. Für unverlangt eingesandte Manuskripte und Listings wird keine Haftung übernommen.

Verlagsleitung: Wolfram Höfler
Operation Manager: Michael Koeppe

Layout: Isabell Schröfl, Benno Schmehl
Bildredaktion: Walter Linne (Fotografie); Ewald Standke, Norbert Raab (Spritzgrafik); Werner Nienstedt (Computergrafik)

Anzeigendirektion: Jens Berendsen
Anzeigenleitung: Philipp Schiede (399) – verantwortlich für die Anzeigen
Telefax: 089/4613-775

Anzeigenverwaltung und Disposition: Chris Mark (421)

Auslandsrepräsentation:
Auslandsniederlassungen:
Schweiz: Markt&Technik Vertriebs AG, Kollerstr. 37, CH-6300 Zug, Tel. 0041/42-440550, Telefax 0041/42-415770
USA: M&T Publishing Inc.; 501 Galveston Drive Redwood City, CA 94063, Telefon: (415) 366-3600, Telefax 415-3663923
Österreich: Markt&Technik Ges. mbH, Große Neugasse 28, A 1040-Wien Telefon: 0043/1/5871393O, Telefax: 0043-1-5871 39333
Anzeigen-Auslandsvertretung:
Großbritannien: Smyth Int. Media Representatives, Telefon 0044/81340-5058, Telefax 0044/81341-9602

Israel: Baruch Schaefer, Telefon 00972-3-5562256, Telefax 00972/52/444518
Taiwan: AIM Int. Inc., Telefon 00886-2-7548613, Telefax 00886-2-7548710
Japan: Media Sales Japan, Telefon 0081/33504-1925, Telefax 0081/33596-1709
Korea: Young Media Inc., Telefon 0082-2-7564819, Telefax 0082-2-7575789
Frankreich: CEP France, Telefon 0033/148007616, Telefax 0033/148240202
Italien: CEP Italia, Telefon 0039/24982997, Telefax 0039/24692834
International Business Manager: Stefan Grajer 089/4613-638

Gesamtvertriebsleiter: York v. Heimburg
Vertriebsmarketing: Helmut Pleyer (710)

Vertrieb Handel: Inland (Groß-, Einzel- und Bahnhofsbuchhandel) sowie Österreich und Schweiz: ip Internationale Presse, Ludwigstraße 26, 7000 Stuttgart 1, Tel. 0711/619660

Einzelheft-Bestellung: Markt & Technik Leserservice, CSJ Postfach 140220, 8000 München 5

Verkaufspreis: Das Einzelheft kostet DM 16,–

Produktion: Klaus Buck (Ltg./180), Wolfgang Meyer (Stellv./887);

Druck: SOV Graphische Betriebe, Laubanger 23, 8600 Bamberg

Sonderdruck-Dienst: Alle in dieser Ausgabe erschienenen Beiträge sind in Form von Sonderdrucken zu erhalten. Anfragen an Reinhard Jarczok, Tel. 089/4613-185, Fax 4613-774.

Vorstand: Otmar Weber (Vors.), Bernd Balzer, Dr. Rainer Doll, Lutz Glandt

Direktor Zeitschriften: Michael M. Pauly

Anschrift für Verlag, Redaktion, Vertrieb, Anzeigenverwaltung und alle Verantwortlichen: Markt&Technik Verlag Aktiengesellschaft, Hans-Pinsel-Straße 2, 8013 Haar bei München, Telefon 089/4613-0, Telex 522052, Telefax 089/4613-100

ISSN 0931-8933

Telefon-Durchwahl im Verlag:
Wählen Sie direkt: Per Durchwahl erreichen Sie alle Abteilungen direkt. Sie wählen 089/4613 und dann die Nummer, die in den Klammern hinter dem jeweiligen Namen angegeben ist.

Copyright-Erklärung

Name: ....

Anschrift: ....

Datum: ....

Computertyp: ....

Benötigte Erweiterung/Peripherie: ....

Datenträger: Kassette/Diskette ....

Programmart: ....

Ich habe das 18. Lebensjahr bereits vollendet

...., den ....

....
(Unterschrift)

Wir geben diese Erklärung für unser minderjähriges Kind als dessen gesetzliche Vertreter ab.

...., den ....

....

....

Bankverbindung:

Bank/Postgiroamt: ....

Bankleitzahl: ....

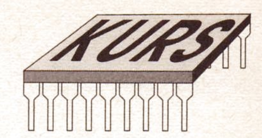
Konto-Nummer: ....

Inhaber des Kontos: ....

Das Programm/die Bauanleitung: ....

das/die ich der Redaktion der Zeitschrift 64'er übersandt habe, habe ich selbst erarbeitet und nicht, auch nicht teilweise, anderen Veröffentlichungen entnommen. Das Programm/die Bauanleitung ist daher frei von Rechten anderer und liegt zur Zeit keinem anderen Verlag zur Veröffentlichung vor. Ich bin damit einverstanden, daß die Markt & Technik Verlag AG das Programm/die Bauanleitung in ihren Zeitschriften oder ihren herausgegebenen Büchern abdruckt und das Programm/die Bauanleitung vervielfältigt, wie beispielsweise durch Herstellung von Disketten, auf denen das Programm gespeichert ist, oder daß sie Geräte und Bauelemente nach der Bauanleitung herstellen läßt und vertreibt bzw. durch Dritte vertreiben läßt.

Ich erhalte, wenn die Markt & Technik Verlag AG das Programm/die Bauanleitung druckt oder sonst verwertet, ein Pauschalhonorar.



spricht hier von einer absolut-X-indizierten Indizierung (bei unserem Poster »Abs,X«).

Das Assembler-Wort LDA haben wir schon oft gehört. Bei unserem Beispiel soll der Akku den Wert aus der Speicherstelle holen, die sich durch $1500 plus dem Inhalt des x-Registers ergibt. Steht im x-Register also zum Zeitpunkt des Befehlsaufrufs »$05«, dann wird der Inhalt aus Speicherstel-ler $1500+$05, also aus $1505 geladen. Da das x-Register Werte zwischen 0 und 255 annehmen kann, können wir allein durch Änderung des x-Registers einen Wert von $1500 bis $15FF indizieren – bei unserem Beispiel in den Akku übernehmen. Mit dieser Adressierung lassen sich plötzlich fantastische Dinge machen: Tabellen vergleichen oder in andere Speicherbereiche schieben usw. Geben Sie dazu im SMON ein:

```
A2000
2000 LDX #00
2002 LDA A09E,X
2005 STA 0400,X
2008 DEX
2009 BNE 2002
200A BRK
200B F
```

und starten Sie mit G2000. Falls Ihr Bildschirm nicht gescrollt ist, sehen Sie am oberen Bildschirmrand 6½ Zeilen Zeichen. Mit <CBM SHIFT> auf Kleinschreibung umgeschaltet, erkennen Sie etliche Befehlswörter aus Basic. Der letzte Buchstabe ist jeweils invertiert. Was haben wir angestellt? Gehen wir einmal den Programmweg durch: In Zeile 2000 wird das x-Register mit $00 geladen und in Zeile 2002 des Akku aus Speicherstelle $A09E+$00 (x-Register) geladen. Was Sie vielleicht nicht wußten: Ab dieser Speicherposition befindet sich im Interpreter eine Liste der Basic-Befehlswörter – den ersten Buchstaben davon haben wir gerade in den Akku geladen. Zeile 2005 bringt den Akku dazu, seinen Inhalt in die Speicherstelle $0400+$00 (x-Register) abzulegen; und hier befindet sich natürlich der Bildschirmspeicher. In Zeile 2008 passiert etwas mit dem x-Register – es wird um »1« verringert. Damit enthält es nicht nicht mehr $00, sondern »$FF«. Würde jetzt die Negativ-Flagge (oder Carry, wenn vorher gesetzt) überprüft, wäre der ganze Vorgang bereits beendet. Aber das geschieht natürlich nicht:

Wir haben bestimmt, daß die Zero-Flagge überprüft wird, und zwar, ob Sie nicht gesetzt ist (BNE). Das kann sie nicht sein, da $FF im x-Register steht, also verzweigt der Mikroprozessor an die Position $2002. Dort lädt er diesmal den Wert aus Speicherstelle $A09E+$FF (=$A19D) in seinen Akku, speichert ihn in Speicherstelle $0400+$FF (=$04FF). Dann verringert er den Wert des x-Registers, überprüft, ob er endlich $00 ist – nein, also zurück zu $2002 und alles solange wiederholt, bis das x-Register gleich $00 ist. Danach erfolgt das wohlverdiente BRK.

Wir ersehen aus dieser Routine, daß auch noch andere Befehle (im Beispiel STA) x-indiziert adressiert werden können. Selbst das y-Register kann »absolut x-indiziert« geladen werden (LDY $$$$,X). Aber Achtung, ein »Abs,X«-speichern dieses Registers ist nicht möglich. Unser obiges Beispiel läßt sich auch umschreiben: man kann auch absolut y-indizieren (LDA $$$$,Y und STA $$$$,Y). Hier gilt die gleiche Besonderheit: Das x-Register läßt sich zwar absolut y-indiziert laden aber so nicht speichern. Und da wir gerade bei Besonderheiten sind: INC und DEC (Erhöhen bzw. Erniedrigen einer Speicherstelle) läßt zwar diese Indizierungsart zu, aber nur mit dem x-Register (s.a. Poster S.26/27).

Unterprogramme – JSR, RTS

Bei unserem Beispiel haben wir zwar ziemlich viele Befehlswörter auf den Bildschirm gebracht. Aber es ist ziemlich mühsam, die einzelnen voneinander zu unterscheiden. Daher verschieben wir im nächsten Beispiel nicht eine komplette 256Byte lange Seite (Page), sondern nur den Bereich in der Länge des ersten Wortes.

```
2000 LDY #00
2002 LDA #0D
2004 JSR FFD2
2007 LDA A09E,Y
200A JSR FFD2
200D SEC
200E INY
200F CPY #03
2011 BNE 2007
2013 BRK
2014 F
```

Diesmal haben wir ein paar (noch) unverständliche Befehle mitverwendet. Bis Zeile 2002 (LDA #0D) kennen wir uns noch aus, aber »JSR FFD2« sollte näher erläutert werden. Mit JSR (Jump SubRoutine – springe in ein Unterprogramm) teilen wir dem Mikroprozessor mit: Merke dir an welcher Stelle du dich gerade befindest und springe an die im Argument angegebene Speicherposition. Der Mikroprozessor legt demzufolge die Position des Programm-Counters auf seinen Stack und ändert den Eintrag im Programm-Counter, in unserem Beispiel auf die Position $FFD2. Danach deutet dieser nicht mehr auf $200D, sondern auf die neue Adresse. Da nach dieser Anweisung der Befehl zu Ende ist, wird das Programm ab Speicherposition $FFD2 fortgesetzt.



Dieser Befehl versetzt uns also in die Lage, aus unserem Programm heraus an eine beliebige andere Speicherposition zu springen; und er macht noch mehr: er merkt sich die Stelle, von der er aus gesprungen ist. In unserem Fall steht ab $FFD2 ein Teil der Betriebsystem-Routinen - die Character-out-Routine (Buchstabenausgabe). Sie wertet das im Akku befindliche Zeichen nach seinem ASCII-Code aus. Die Ausgabe beginnt ab Cursor-Position.

Wir haben $0D (13) geladen. Dieser Wert ist die ASCII-Kodierung für einen Zeilenvorschub. Das heißt nach dem ersten Aufruf dieser Routine befinden wir uns für die Ausgabe des nächsten Zeichens am Zeilenanfang, eine Zeile tiefer. Dieses erste $0D ist deshalb wichtig, da sonst die folgenden Ausgaben unmittelbar nach oben erwähnter Null beginnen. Unser Mikroprozessor ist also nach $FFD2 gesprungen und führt dort die Ausgaberoutine aus. Aber wie kommt er wieder zurück?

Wir haben früher schon einmal von dem dafür zuständigen Befehl gehört - RTS (ReTurn from Subroutine, kehre vom Unterprogramm zurück). Dieser Befehl muß hinter dem letzten Befehl des angesprungenen Programmteils stehen. In der Character-out-Routine ist das der Fall. Ein RTS ändert nur den Programm-Counter, indem sein Befehlscode dem Mikroprozessor mitteilt: nimm die beiden obersten Werte vom Stack und schiebe sie in den PC. Damit ist dein Befehl beendet. Falls wir nun zwischenzeitlich nichts am Stack geändert haben, springt die Befehlsausführung zu dem Programmschritt zurück, von dem aus die Unterroutine aufgerufen wurde.

Allein an dieser Beschreibung haben Sie schon gemerkt, daß man die Rücksprungadresse manipulieren kann, doch Vorsicht: willkürliches Ändern hat in den meisten Fällen den Absturz der CPU zur Folge.

Sehen wir uns unsere Routine weiter an:

Befehlswort	Adressierung	Bytezahl	Code Hex	Code Dez	Taktzyklen	Beeinflussung von Flaggen
AND	absolut	3	2D	45	4	N, Z
	0-page-abs	2	25	37	3	N, Z
	unmittelbar	2	29	41	2	N, Z
	abs.-X-indiz.	3	3D	61	4 *	N, Z
	abs.-Y-indiz.	3	39	57	4 *	N, Z
	indiz.-indir.	2	21	33	6	N, Z
	indir.-indiz.	2	31	49	5 *	N, Z
	0-page-X-indiz	2	35	53	4	N, Z
ORA	absolut	3	0D	13	4	N, Z
	0-page-abs.	2	05	05	3	N, Z
	unmittelbar	2	09	09	2	N, Z
	abs.-X-indiz.	3	1D	29	4 *	N, Z
	abs.-Y-indiz.	3	19	25	4 *	N, Z
	indiz.-indir.	2	01	01	6	N, Z
	indir.-indiz.	2	11	17	5 *	N, Z
	0-page-X-indiz	2	15	21	4	N, Z
EOR	absolut	3	4D	77	4	N, Z
	0-page abs.	2	45	69	3	N, Z
	unmittelbar	2	49	73	2	N, Z
	abs.-X-indiz.	3	5D	93	4 *	N, Z
	abs.-Y-indiz.	3	59	89	4 *	N, Z
	indiz.-indir.	2	41	65	6	N, Z
	indir.-indiz.	2	51	81	5 *	N, Z
	0-page-X-indiz	2	55	85	4	N, Z
ASL	»Akkumulator«	1	0A	10	2	N, Z, C
	absolut	3	0E	14	6	N, Z, C
	0-page-abs.	2	06	06	5	N, Z, C
	abs.-X-indiz.	3	1E	30	7	N, Z, C
	0-page-X-indiz	2	16	22	6	N, Z, C

* bedeutet: Bei seitenüberschreitenden Indizierungen muß noch ein Taktzyklus dazugerechnet werden.

Die logischen Operationen AND, ORA und EOR auf einen Blick. ASL finden Sie auf Seite 24.

Wir befinden uns mittlerweile am Anfang der Zeile nach G2000. Falls dies zufällig die letzte Bildschirmzeile war, hat die Character-out-Routine dafür gesorgt, daß der Bildschirm gescrollt hat. In Zeile 2000 sorgte der Befehl LDY #00 für ein Laden des Werts $00 in das y-Register. In Zeile 2007 wird $A09E, y geladen und danach die Character-out-Routine wieder aufgerufen. Kurz eine Erklärung zu dem »e« ($45), das sich gerade im Akku befindet. Wir haben im ersten Beispiel dieses »e« genommen und einfach in den Bildschirmspeicher übertragen. Der Dank dafür war, daß wir ohne Umschaltung auf Kleinschrift einen Strich an der ersten Bildschirmposition hatten. Nach dem Umschalten wurde er ein großes »E«. Der Grund für diese ungewöhnliche Reaktion: die Basic-Befehlswörter sind im ASCII-Code gespeichert. Wir hatten aber diesen Code direkt in den Bildschirmspeicher geschrieben und dabei Glück, daß überhaupt etwas erkennbar war - denn Bildschirm- und ASCII-Code stimmen nicht überein. Auf jeden Fall verwenden wir diesmal die richtige Methode und geben über $FFD2 das ASCII-Zeichen aus. Wir erhöhen jetzt das y-Register um »1« und vergleichen mit $03 (weil wir wissen, daß es drei Ziffern sind). Ergibt dieser Vergleich nicht Null, wird das nächste Zeichen ausgegeben, ansonsten geht's mit BRK zurück zum SMON.

Logische Operationen– AND, ORA, EOR und BIT

Von den Basic-Befehlsworten haben wir vorhin das erste Wort angezeigt, aber nur weil wir wußten, daß es drei Buchstaben besitzt. Das ist einigermaßen unergiebig, da ja alle Befehle unterschiedlich lang sind. Also sollten wir uns eine Methode überlegen, bei der dieser Unterschied berücksichtigt wird:

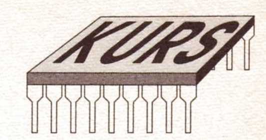
```
A 200A
200A AND #80
200C BEQ 201A
200E LDA A09E,Y
2011 AND #7F
2013 JSR FFD2
2016 INY
2017 BNE 2002
2019 BRK
201A LDA A09E,Y
201D INY
201E BNE 2004
2020 BRK
2021 F
```

Nach diesen Zeilen sollten Sie D20002021 eingeben, um die gesamte Routine zu sehen. Bis Zeile 200A sind wir mit dem vorigen Beispiel identisch, doch hier erscheint ein neuer Befehl: AND. Er führt eine logische UND-Verknüpfung des Akkus mit dem Argument hinter AND durch (bitweise). Das Ergebnis steht anschließend im Akku. Um die Wirkung dieses Befehls zu durchleuchten, betrachten wir uns einmal die gerade verwendeten Zahlen in Binärdarstellung:

```
Akku $45 %0100101
AND  $80 %1000000
-----------------
Erg. $00 %0000000
```

AND wirkt als bitweises Filter. Es läßt nur dann ein Bit durch, wenn sowohl im Akku als auch im Argument an der gleichen Bit-Position eine »1« steht. Dann erscheint als Ergebnis im Akku ebenfalls eine »1« an dieser Stelle. Anders ausgedrückt:

```
0 AND 0 ergibt 0
0 AND 1 ergibt 0
1 AND 0 ergibt 0
1 AND 1 ergibt 1
```



Von diesen vier möglichen Zuständen der sich entsprechenden Bits ergibt nur 1 AND 1 das logische Ergebnis 1. Sehen wir uns daher die Basic-Wort-Tabelle an; »MA09E A0A6« zeigt die ersten acht Inhalte und daneben die Zeichen (auf Groß/Kleinschrift umschalten!):

```
:a09e 45 4e c4 46 4f d2 4e 45   enDfoRne
```

Allein am Text rechts sieht man schon, daß mit dem »D« von enD und mit dem »R« von foR etwas geschehen sein muß; beide erscheinen in Großbuchstaben. Der Trick dieser Tabelle basiert auf der Tatsache, daß bei Großbuchstaben des ASCII-Satzes grundsätzlich Bit 7 gesetzt ist. »d« hat daher den Wert $44 (%01000100) und »D« entsprechend $c4 (%11000100). Unsere Methode, dies zu unterscheiden, war die AND-Verknüpfung mit $80 (%10000000). Das Ergebnis dieser AND-Verknüpfung kann bei uns nur dann ein höherer Wert als Null sein, wenn bei einem Wert der getesteten Tabelle Bit 7 gesetzt, also dieser Wert größer 128 ist. Dann ist auch im Akku dieses Bit gesetzt und damit die Zero-Flagge gelöscht.

Der BEQ-Befehl zwingt den Mikroprozessor dazu, bei gesetzter Zero-Flagge nach $201a zu verzweigen. Beim dritten Buchstaben aber ist Bit 7 gesetzt, die Zero-Flagge wird gelöscht und unser Programm wird ab $200E fortgeführt. Hier laden wir nochmal den letzten Wert in den Akku (wir hatten ihn ja mit AND verändert) und führen wieder eine AND-Verknüpfung durch; diesmal mit $F7 (%01111111). Der Effekt: Bit 7 wird gelöscht:

```
Akku $C4 %11000100
AND  $7F %01111111
------------------
Erg. $44 %01000100
```

Wir können also mit AND nicht nur testen, sondern gezielt auch Bits löschen. Als nächsten Schritt (Zeile 2013) senden wir diesen Wert zur Character-out-Routine. Sie muß ein »d« ausgeben, da ja Bit 7 gelöscht und die Zahl damit kleiner als 128 ist. In der Folge erhöhen wir das y-Register und überprüfen, ob es bereits Null ist. Wenn nicht, beginnt die Routine einfach mit einem Zeilenrücklauf ab Zeile 2002 aufs neue. Damit beginnt das nächste Befehlswort in der nächsten Zeile. Im anderen Falle BRK.

Betrachten wir noch kurz den Programmteil ab Zeile 201A: Auch hier wird das entsprechende Zeichen aus der Tabelle geladen, dann das y-Register um eins erhöht und auf Null getestet (Null = BRK). Ist es jedoch nicht Null, verzweigt das Programm einfach zur Zeile 2004, in der die CHROUT-Routine aufgerufen wird (das Zeichen ist ja noch im Akku).

Wir könnten uns übrigens einiges an Programmlänge sparen, wenn wir den AND in Zeile 200A nicht verwenden und anschließend anstelle, von BEQ, über die Negativ-Flagge verzweigen (BPL - Branch if PLus). Eine andere (um ein Byte längere) Methode ist die Carry-Flagge zu setzen und mit dem Befehl »CMP #127« zu überprüfen, ob der Wert größer/gleich 128 ist. CMP #128 genügt dabei nicht, da 128 im Akku mit 128 verglichen (-128) Null ergibt, und damit die Carry-Flagge noch nicht löscht (kein Übertrag).

Starten Sie (mit G2000) jetzt einfach die Routine. Sie sehen die Basic-Befehle, die in den ersten 256 Bytes der Basic-Befehlstabelle enthalten sind, im Klartext.

Als nächsten logischen Befehl lernen wir ORA kennen (inclusive OR with Akkumulator - ODER-Verknüpfung eines Arguments mit dem Akku). Mit dieser Funktion lassen sich gezielt Bits setzen (kennen Sie sicher schon aus Basic). Hier wird der Akku bitweise mit dem Argument verknüpft. Das Ergebnis liegt wieder im Akku vor. Dabei gilt folgende Wahrheitstabelle:

```
0 ORA 0 = 0
1 ORA 0 = 1
0 ORA 1 = 1
1 ORA 1 = 1
```

Wir haben drei von vier der möglichen Bit-Kombinationen, bei denen das Ergebnis-Bit gesetzt wird. Dazu drei Beispiele:

```
Akku $C4 %11000100
ORA  $00 %00000000
------------------
Erg. $C4 %01000100
```

Eine ODER-Verknüpfung mit Null ändert am Akku-Inhalt nichts.

```
Akku $C4 %11000100
ORA  $68 %01100010
------------------
Erg. $E6 %11100110
```

Gesetzte Bits des Akku erscheinen genauso wie gesetzte Bits des Arguments im Ergebnis. Interessant ist diese Funktion zum Setzen von Bits in der Bit-Map (Grafikmodus des C64). Die anderen Bits werden nicht beeinflußt.

```
Akku $44 %01000100
ORA  $80 %10000000
------------------
Erg. $C4 %11000100
```

Dieses Beispiel stellt die Umkehrfunktion zu der im Beispiel-Listing benützten UND-Verknüpfung dar.

EOR (Exclusive-OR - Exclusive ODER-Verknüpfung) haben wir schon beim Subtrahieren gestreift. Dieser Befehl invertiert die sich gegenüberstehenden gleichen Bits. Ungleiche Bits werden nicht geändert. Das Ergebnis ist wieder im Akku:

```
0 EOR 0 = 0
1 EOR 0 = 1
0 EOR 1 = 1
1 EOR 1 = 0
```

oder als Zahlenbeispiel:

```
Akku $44 %01000100
EOR  $C2 %11000010
------------------
Erg. $BF %10111111
```

Eine Anwendung haben wir schon kennengelernt, zusätzlich läßt sich mit dieser Funktion z.B. eine Bit-Map invertieren. Im Betriebssystem wird mit dieser Funktion das Cursor-Blinken erzeugt (Bildschirm-Code für Space ist $20).

```
Bsch $20 %00100000 (Space)
EOR  $80 %10000000
------------------
Bsch $A0 %10100000 (Space invertiert)
```

Beim nächsten Aufruf der Routine erscheint:

```
Bsch $A0 %10100000 (Space invertiert)
EOR  $80 %10000000
------------------
Bsch $20 %00100000 (Space)
```

Zwischen diesen Invertierungen muß verzögert werden. Der Wechsel geschähe sonst so schnell, daß nur ein Flimmern sichtbar wäre. Bei der Routine für den Cursor passiert dies im Interrupt. Jede 60stel Sekunde werden Zähler erhöht. Wenn das Low-Byte einen bestimmten Wert erreicht, wird einfach die Invertierungs-Routine aufgerufen.

Bleibt nur noch ein logischer Befehl: BIT (test BITs - prüfe die Bits). Dieser Befehl führt eine UND-Verknüpfung der Bits des Akkus mit dem Argument durch. Das Ergebnis erscheint allerdings nicht im Akku, sondern in der Negativ- (Bit 7) und der Overflow-Flagge (Bit 6). Sowohl Akku als auch Argument behalten ihren ursprünglichen Wert. Das Ergebnis der Verknüpfung der Bits 0 bis 5 wird zwar nicht festgehalten, wohl gibt aber die Zero-Flagge Auskunft darüber, es Null (Z=1) oder größer Null war (Z=0). Bei geeigneter Maskenwahl kann also über die drei Flaggen jedes Bit getestet werden:

```
Akku $C4 %11000100
BIT  $44 %01000100
------------------
Erg. $44 %01000100 (wird nicht festgehalten)
Negativ-Flagge   (N) = 0
Overflow-Flagge  (V) = 1
Zero-Flagge      (Z) = 0
```

Wenn wir zwei andere Zahlen verwenden, wird auch die Funktion der Zero-Flagge sichtbar:

```
Akku $C4 %11000100
BIT  $80 %10000000
------------------
Erg. $44 %10000000 (wird nicht festgehalten)
Negativ-Flagge   (N) = 1
Overflow-Flagge  (V) = 0
Zero-Flagge      (Z) = 0
```

und als zweite Zahl

```
Akku $C4 %11000100
BIT  $20 %00100000
------------------
Erg. $00 %00000000 (wird nicht festgehalten)
Negativ-Flagge   (N) = 0
Overflow-Flagge  (V) = 0
Zero-Flagge      (Z) = 1
```

Bei der Verwendung von BIT sollten Sie berücksichtigen, daß dieser Befehl nur die Adressierungsarten Absolut (BIT $$$$) und Zero-Page (BIT $$) beherrscht. Sie müssen das Argument also in einer Speicherstelle parat haben. Falls Sie eine Maske wollen, läßt sich dies mit LDA #$$ bewerkstelligen.

Befehlswort	Adressierung	Bytezahl	Code Hex	Code Dez	Taktzyklen	Beeinflussung von Flaggen
LSR	»Akkumulator«	1	1A	26	2	N,Z,C
	absolut	3	4E	78	6	N,Z,C
	0-page-absolut	2	46	70	5	N,Z,C
	absolut-X-indiz.	3	5E	94	7	N,Z,C
	0-page-X-indiz.	2	56	86	6	N,Z,C
ROL	»Akkumulator«	1	2A	42	2	N,Z,C
	absolut	3	2E	46	6	N,Z,C
	0-page-absolut	2	26	38	5	N,Z,C
	absolut-X-indiz.	3	3E	62	7	N,Z,C
	0-page-X-indiz.	2	36	54	6	N,Z,C
ROR	»Akkumulator«	1	6A	106	2	N,Z,C
	absolut	3	6E	110	6	N,Z,C
	0-page-absolut	2	66	102	5	N,Z,C
	absolut-X-indiz.	3	7E	126	7	N,Z,C
	0-page-X-indiz.	2	76	118	6	N,Z,C

Drei Befehle zum bitweisen Verschieben

Verschiebebefehle – ASL, ROL, LSR und ROR

Sie sind mittlerweile fast Programmierprofi geworden, doch einige mathematische Befehle sollten Sie noch kennenlernen. Die Basis aller Rechnungen des Mikroprozessors ist ein Byte. Ein Byte besteht aus acht Bit. Jedes höherwertige Bit hat die doppelte Wertigkeit des jeweils niedrigerwertigen. Das schauen wir uns einmal genauer an:

%00000001	%00000010	%00000100	%00001000
1	2	4	8

Was haben wir bei unserem Beispiel gemacht? Wir haben, um von 1 auf »2« zu kommen, das Bit um eine Stelle nach links verschoben. Eine weitere Verschiebung nach links macht aus »2« eine »4«. Das heißt eine Stellenverschiebung nach links entspricht einer Multiplikation mit zwei.

Versuchen wir dieses Rechenbeispiel mit anderen Zahlen:

```
%00001011 (dez. 11)
%00010110 (dez. 22)
```

Auf diese Art lassen sich also auch größere Zahlen verdoppeln - und, natürlich läßt sich dieser Vorgang auch Umdrehen. Das heißt, eine Stellenverschiebung nach rechts entspricht einer Division durch zwei.

Unser Mikroprozessor beherrscht diese Rechenarten:

ASL (Arithmetik Shift Left - rechnerische Linksverschiebung) verschiebt den Akku oder die adressierte Speicherstelle um ein Bit nach links, in das nullte Bit wird eine »0« geschoben. Wäre das alles, ließe sich nur mit einem Byte rechnen. Das siebente Bit, das jetzt nicht mehr ins Byte hineinpaßt, bedeutet einen Übertrag. Und richtig, es wird in die Carry-Flagge geschoben:

```
A5000
5000 LDA #8B (%10001011)
5002 ASL
5003 BRK
5004 F
```

ergibt nach dem Start (G5000) im Akku $16 (%00010110) und eine gesetzte Carry-Flagge.

Um ein höherwertiges Byte damit zu verknüpfen, müssen wir dieses zuerst um ein Bit verschieben, dann die Carry-Flagge in die niedrigste Stelle bringen. Dafür ist ein anderer Befehl zuständig:

ROL (ROtate Left one bit - verschiebe ein Bit nach links). Bei ihm wird zuerst um ein Bit nach links geschoben, das siebte (jetzt achte) Bit gemerkt und die Carry-Flagge in die nullte Stelle geschoben. Der gemerkte Übertrag wandert wieder in die Carry-Flagge. Wozu braucht man in der Praxis diesen Befehl?

Nehmen wir an, Sie haben im Speicher ab Speicherposition $4000 eine Tabelle angelegt, jeweils im Low- / High-Byte-Format. Sie wollen von dieser Tabelle die beiden Bytes laden, die an 129ster Stelle liegen. Da Ihre Tabelle je aus 2Byte besteht, benötigen Sie also den (129 x 2=258) 258sten und 259sten Wert, und zwar ab Speicherstelle $4000. Wir gehen folgendermaßen vor:

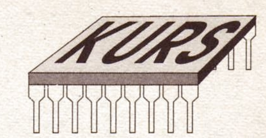
```
A3A00
3A00 LDA #81
3A02 STA  A8
3A04 LDA #00
3A06 STA  A9
3A08 ASL  A8
3A0A ROL  A9
3A0C CLC
3A0D LDA  A9
3A0F ADC #40
3A11 STA  A9
3A13 BRK
3A14 F
```

In Zeile 3A00 wird 129 ($81) geladen und in Speicherstelle $A8 abgelegt. Da 129 in ein Byte paßt, kommt in die zweite Speicherstelle ($A9) der Wert Null. In Zeile 3A08 wird das Low-Byte mit zwei multipliziert, der Übertrag ist in der Carry-Flagge, und in Zeile 3A0A wird das High-Byte mit zwei multipliziert und der Inhalt der Carry-Flagge in das niedrigstwertige Bit übernommen. Anschließend wird zum High-Byte $40 addiert. Wenn Sie jetzt mit G3A00 starten und sich mit M00A8 Low- und High-Byte betrachten, steht dort:

```
:00A8 02 41 usw.
```

Damit ist der richtige Wert in den Speicherstellen. Man kann diese als Pointer verwenden. Doch zuvor noch zwei andere Befehle:

LSR (Logical Shift Right - logisches Verschieben nach rechts) verschiebt den Inhalt des Bytes um ein Bit nach rechts, und Bit 7 wird Null. Bit 0 kommt in die Carry-Flagge.



LSR ist die Umkehrung von ASL.

ROR (ROtate Right one bit - verschiebe um ein Bit nach rechts) verschiebt den Inhalt des Bytes um ein Bit nach rechts und bringt die Carry-Flagge in Bit 7. Auch hier wird Bit 0 in die Carry-Flagge übertragen. ROR ist die Umkehrung von ROL.

Die indirekt-indizierte und die indiziert-indirekte Adressierung

Wir sagten, daß unsere Werte als Pointer verwendet werden können. Pointer heißt »Zeiger« und hat etwas mit indirekt-indizierter Adressierung zu tun. Vervollständigen wir unser Programmbeispiel:

```
A3A13
3A13 LDY #00
3A15 LDA (A8),Y
3A17 STA 4000,Y
3A1A INY
3A1B SEC
3A1C CPY #04
3A1E BNE 3A15
3A20 BRK
3A21 F
```

Zeile 3A13 lädt das y-Register mit dem Wert Null. Das ist uns wohl bekannt. Aber Zeile 3A15 ist neu für uns. Hier wird nicht, wie man zunächst vermuten könnte, aus $A8 plus Y geladen, sondern in $A8 und $A9 steht ein Zeiger (Pointer) auf eine Adresse. In unserem Fall (durch die vorherige Rechnung) steht in $A8 »02« und in $A9 »41«. Damit deutet der Zeiger auf die Speicherstelle $4102. Wir laden in unserem Fall den Wert aus der Speicherstelle $4102 + Y (=0) und speichern ihn in die Zelle $4000 + Y. Danach erhöhen wir das y-Register, vergleichen mit $04 und wiederholen die Aktion: Diesmal holen wir aus Speicherstelle $4102 + $01 = $4103 und speichern in $4001. Der Vorgang wird solange wiederholt, bis im y-Register die Zahl »04« steht, dann erfolgt BRK. Damit haben wir vier Werte übernommen und auf einen anderen Bereich übertragen. Sie sehen den Vorteil dieser »indirekt-indizierten« Adressierung. Durch sie lassen sich die Speicherpositionen berechnen, aus denen Werte geholt und bearbeitet werden. Zwei Dinge sind dabei zu beachten:

1. Indirekt-indiziert läßt sich nur auf die Zero-Page anwenden.
2. Diese Adressierung läßt nur das y-Register zu.

Für das x-Register steht eine andere Indizierungsart parat: die indiziert-indirekte Adressierung. Ihre Schreibweise:

```
LDA (A9,X)
```

hat also eine gewisse Ähnlichkeit. Der angegebene Wert (A9) muß wieder eine Zero-Page-Adresse sein. Der Wert in der Speicherstelle, auf die »indiziert« wird, dient als High-Byte und das x-Register als Low-Byte. Bei uns steht in $A9 der Wert $41. Wenn das x-Register auf $00 steht, wird oben aus Speicherstelle $4100 geladen. Hat das x-Register $01 zum Inhalt, wird $4101 übernommen usw.

Welche Befehle für welche Adressierungsarten erlaubt sind, sehen Sie auf unserem Poster auf S. 26/27.

Eine indirekte Adressierung soll hier nicht vergessen werden: der indirekte Sprung. Wir hatten bei JSR gehört, daß, zu einem Unterprogramm gesprungen, vorher die Position des Mikroprozessors gemerkt wird und RTS wieder an die ursprüngliche Position zurückkehrt. Der Befehl JMP (JuMP to adress - springe zu einer Adresse) ist die dem »GOTO«-BASIC-Befehl entsprechende Anweisung in Assembler. Hier wird ohne Rücksicht auf die derzeitige Position auf die Adresse verzweigt, die als Argument dient.

```
JMP 4000
```

springt nach $4000. Dieser Befehl ist aber auch indirekt anwendbar. Für unser Beispiel könnte man schreiben:

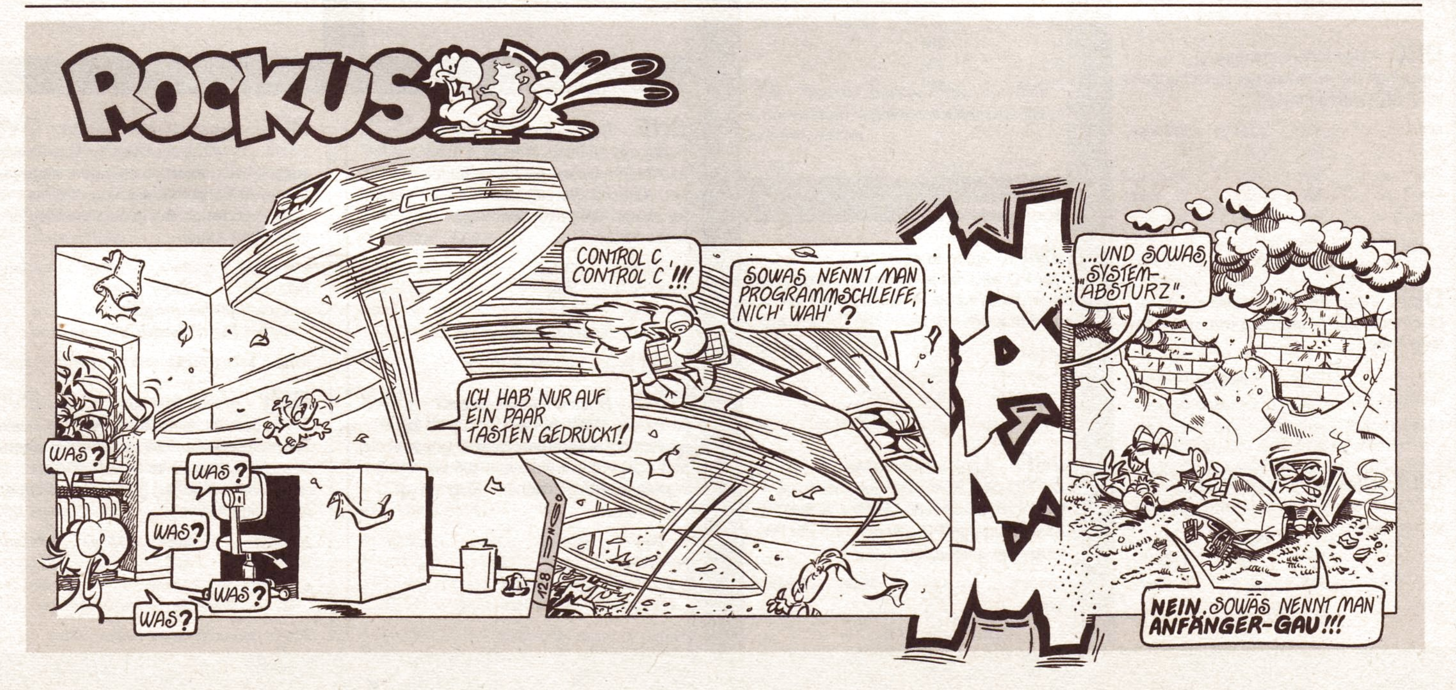
```
JMP (00A8)
```

Damit Sie den indirekten Sprung kennenlernen, befindet sich Listing 14 auf Diskette. Laden und betrachten Sie es sich (L″LI.14″ und D1400140A).

Transportbefehle im Mikroprozessor – TXA, TAX, TYA, TAY, TSX und TXS

Ab und zu ist es nötig, Registerinhalte gegeneinander auszutauschen. Viele Dinge können nur im Akku geschehen (Addition, Subtraktion usw.). Wollen Sie eine dieser Operationen z.B. mit dem x-Register durchführen, verschieben Sie einfach zuerst den x-Inhalt in den Akku mit TXA (Transfer register X into Accumulator - kopiere den Inhalt des x-Registers in den Akku). Dann führen Sie die Operation durch und kopieren wieder den Akku zurück ins x-Register mit TAX (Transfer Accumulator into register X - kopiere den Akku ins x-Register). Das gleiche läßt sich mit dem y-Register bewerkstelligen:

TYA (Transfer register Y into Accumulator - kopiere y-Register in den Akku) und TAY (Transfer Accumulator into register Y - kopiere Akku ins y-Register).



6510: alle Befehle

Verschiebebefehle

ASL – Arithmetic shift left

Akkumulator oder Speicherstelleninhalt um ein Bit nach links verschieben. Bit 0 wird gelöscht, Bit 7 ins C-Flag geschoben. Das Ergebnis steht in der Datenquelle.

Adr.Art	Code	Länge	Zyklen
Akku	$0a	1	2
Abs:	$0e	3	6
OPge	$06	2	5
Abs.X	$1e	3	7
OPge,X	$16	2	6

Flags: NVBDIZC
* **

LSR – Logical shift right

Akkumulator oder Speicherstelleninhalt um ein Bit nach rechts verschieben. Bit 7 wird gelöscht, Bit 0 ins C-Flag geschoben. Das Ergebnis steht in der Datenquelle.

Adr.Art	Code	Länge	Zyklen
Akku	$4a	1	2
Abs:	$4e	3	6
OPge	$46	2	5
Abs.X	$5e	3	7
OPge,X	$56	2	6

Flags: NVBDIZC
0 **

ROL – Rotate left one bit

Akkumulator oder Speicherstelleninhalt wird um ein Bit nach links verschoben. C-Flag-Inhalt nach Bit 0 und Bit 7 ins C-Flag.

Adr.Art	Code	Länge	Zyklen
Akku	$2a	1	2
Abs:	$2e	3	6
OPge	$26	2	5
Abs.X	$3e	3	7
OPge,X	$36	2	6

Flags: NVBDIZC
* **

ROR – Rotate right one bit

Akkumulator oder Speicherstelleninhalt wird um ein Bit nach rechts verschoben. C-Flag-Inhalt nach Bit7 und Bit0 ins C-Flag.

Adr.Art	Code	Länge	Zyklen
Akku	$6a	1	2
Abs:	$6e	3	6
OPge	$66	2	5
Abs.X	$7e	3	7
OPge,X	$76	2	6

Flags: NVBDIZC
* **

Arithmetik

INC – Increment memory

Der Inhalt der adressierten Speicherstelle wird um 1 inkrementiert

Adr.Art	Code	Länge	Zyklen
Abs:	$ee	3	6
OPge	$e6	2	5
Abs.X	$fe	3	7
OPge,X	$f6	2	6

Flags: NVBDIZC
* *

INX – Increment X-Register

Der Inhalt des X-Registers wird um 1 inkrementiert

Adr.Art	Code	Länge	Zyklen
Implizit	$e8	1	2

Flags: NVBDIZC
* *

INY – Increment Y-Register

Der Inhalt des Y-Registers wird um 1 inkrementiert.

Adr.Art	Code	Länge	Zyklen
Implizit	$c8	1	2

Flags: NVBDIZC
* *

DEC – Decrement memory

Der Inhalt der adressierten Speicherstelle wird um 1 dekrementiert

Adr.Art	Code	Länge	Zyklen
Abs:	$ce	3	6
OPge	$c6	2	5
Abs.X	$de	3	7
OPge,X	$d6	2	6

Flags: NVBDIZC
* *

DEX – Decrement X-Register

Der Inhalt des X-Registers wird um 1 dekrementiert

Adr.Art	Code	Länge	Zyklen
Implizit	$ca	1	2

Flags: NVBDIZC
* *

DEY – Decrement Y-Register

Der Inhalt des Y-Registers wird um 1 dekrementiert.

Adr.Art	Code	Länge	Zyklen
Implizit	$88	1	2

Flags: NVBDIZC
* *

Übertragen in Speicher

STA – Store accumulator in memory

Der Inhalt des Akku wird in die adressierte Speicherstelle geschrieben.

Adr. Art	Code	Länge	Zyklen
Abs:	$8d	3	4
OPge	$85	2	3
Abs.X	$9d	3	5
Abs.Y	$99	3	5
(Ind.X)	$81	2	6
(Ind),Y	$91	2	6
OPge,X	$95	2	4

Flags: keine Veränderung

STX – Store register X in memory

Der Inhalt des X-Register wird in die adressierte Speicherstelle geschrieben.

Adr.Art	Code	Länge	Zyklen
Abs:	$8e	3	4
OPge	$86	2	3
OPge,Y	$96	2	4

Flags: keine Veränderung

STY – Store register Y in memory

Der Inhalt des Y-Register wird in die adressierte Speicherstelle geschrieben.

Adr.Art	Code	Länge	Zyklen
Abs:	$8c	3	4
OPge	$84	2	3
OPge,X	$94	2	4

Flags: keine Veränderung

Sprungbefehle

JMP – Jump to address

Der PC wird mit einer neuen Adresse geladen, was zu einem Sprung im Programm führt.

Adr.Art	Code	Länge	Zyklen
Abs:	$4c	3	3
Indirekt	$6c	3	5

Flags: keine Veränderung

JSR – Jump to subroutine

Der PC plus 2 wird auf den Stapel gebracht. Anschließend wird die neue Adresse in den PC geladen und die Unterroutine aufgerufen.

Adr.Art	Code	Länge	Zyklen
Abs:	$20	3	6

Flags: keine Veränderung

Beeinflussen der Flags

CLC – Clear carry

Das C-Flag wird auf 0 gesetzt.

Adr.Art	Code	Länge	Zyklen
Implizit	$18	1	2

Flags: NVBDIZC
0

SEC – Set carry

Das C-Flag wird auf 1 gesetzt.

Adr.Art	Code	Länge	Zyklen
Implizit	$38	1	2

Flags: NVBDIZC
1

CLV – Clear overflow flag

Das V-Flag wird auf 0 gesetzt.

Adr.Art	Code	Länge	Zyklen
Implizit	$b8	1	2

Flags: NVBDIZC
0

CLD – Clear dezimal mode

Das D-Flag wird auf 0 gesetzt. Die Befehle ADC und SBC arbeiten binär, bis das D-Flag wieder auf 1 gesetzt wird.

Adr.Art	Code	Länge	Zyklen
Implizit	$d8	1	2

Flags: NVBDIZC
0

SED – Set decimal mode

Das D-Flag wird auf 1 gesetzt. Die Befehle ADC und SBC arbeiten dezimal, bis das D-Flag wieder auf 0 gesetzt wird.

Adr.Art	Code	Länge	Zyklen
Implizit	$f8	1	2

Flags: NVBDIZC
1

CLI – Clear interrupt flag

Das I-Flag wird auf 0 gesetzt. Es werden weitere Programmunterbrechungen (Interrupts) zugelassen.

Adr.Art	Code	Länge	Zyklen
Implizit	$58	1	2

Flags: NVBDIZC
0

SEI – Set interrupt mask

Es werden keine weiteren Programmunterbrechungen (Interrupts) zugelassen.

Adr. Art	Code	Länge	Zyklen
Implizit	$78	1	2

Flags: NVBDIZC
1

Addition/Subtraktion

ADC – Add with Carry

Addiert ein Argument zum Akku unter Berücksichtigung des C-Flag.

Besonderheiten:
- arbeitet binär oder dezimal
- für korrektes Addieren muß das C-Flag gelöscht sein

Adr.Art	Code	Länge	Zyklen
Abs:	$6d	3	4
OPge	$65	2	3
imm	$69	2	2
Abs.X	$7d	3	4*
Abs.Y	$79	3	4*
(Ind.X)	$61	2	6
(Ind),Y	$71	2	5*
OPge,X	$75	2	4

* Zuzüglich 1 Zyklus bei Seitenüberschreitung

Flags: NVBDIZC
** **

SBC – Subtract with Carry

Subtrahiert ein Argument vom Akku unter Berücksichtigung des C-Flag.

Besonderheiten:
- arbeitet binär oder dezimal
- für korrektes Addieren muß das C-Flag gesetzt sein

Adr.Art	Code	Länge	Zyklen
Abs:	$ed	3	4
OPge	$e5	2	3
imm	$e9	2	2
Abs.X	$fd	3	4*
Abs.Y	$f9	3	4*
(Ind.X)	$e1	2	6
(Ind),Y	$f1	2	5*
OPge,X	$f5	2	4

* Zuzüglich 1 Zyklus bei Seitenüberschreitung

Flags: NVBDIZC
** **

CPY – Compare to register Y

Die adressierten Daten werden vom Y-Register abgezogen, das Ergebnis jedoch nicht gespeichert. Die Flags N, Z und C werden entsprechend gesetzt. Z = 1, wenn beide Werte gleich sind. N = 1 und C = 0, wenn Y kleiner als die adressierten Daten ist. C = 1, wenn der Inhalt des Y-Register größer oder gleich ist.

Adr.Art	Code	Länge	Zyklen
Abs:	$cc	3	4
OPge	$c4	2	3
imm	$c0	2	2

Flags: NVBDIZC
* **

Verzweigungsbefehle

BNE – Branch if not equal to zero

Testet das Nullflag. Ist Z = 0, wird zur nächsten Adresse plus dem angegebenen Abstand (Bereich von -128 bis +127) verzweigt. Bei gesetztem Flag erfolgt keine Aktion.

Adr.Art	Code	Länge	Zyklen
Relativ	$d0	2	2*

* +1, bei Verzweigung
+2, bei Page-Überschreitung

Flags: keine Veränderung

BEQ – Branch if equal to zero

Testet das Nullflag. Ist Z = 1, wird zur nächsten Adresse plus dem angegebenen Abstand (Bereich von -128 bis +127) verzweigt. Bei gelöschtem Flag erfolgt keine Aktion.

Adr.Art	Code	Länge	Zyklen
Relativ	$f0	2	2*

* +1, bei Verzweigung
+2, bei Page-Überschreitung

Flags: keine Veränderung

BPL – Branch if plus

Testet das Vorzeichenflag. Ist N = 0, wird zur nächsten Adresse plus dem angegebenen Abstand (Bereich von -128 bis +127) verzweigt. Bei gesetztem Flag erfolgt keine Aktion.

Adr.Art	Code	Länge	Zyklen
Relativ	$10	2	2*

* +1, bei Verzweigung
+2, bei Page-Überschreitung

Flags: keine Veränderung

BMI – Branch if minus

Testet das Vorzeichenflag. Ist N = 1, wird zur nächsten Adresse plus dem angegebenen Abstand (Bereich von -128 bis +127) verzweigt. Bei gelöschtem Flag erfolgt keine Aktion.

Adr.Art	Code	Länge	Zyklen
Relativ	$30	2	2*

* +1, bei Verzweigung
+2, bei Page-Überschreitung

Flags: keine Veränderung

auf einen Blick

CMP – Compare to accumulator

Die adressierten Daten werden vom Akku abgezogen, das Ergebnis jedoch nicht gespeichert. Die Flags N, Z und C werden entsprechend gesetzt. Z = 1, wenn beide Werte gleich sind. N = 1 und C = 0, wenn der Akku kleiner als die adressierten Daten ist. C = 1, wenn der Inhalt des Akku größer oder gleich ist.

Adr.Art	Code	Länge	Zyklen
Abs:	$cd	3	4
OPge	$c5	2	3
imm	$c9	2	2
Abs.X	$dd	3	4*
Abs.Y	$d9	3	4*
(Ind.X)	$c1	2	6
(Ind),Y	$d1	2	5*
OPge,X	$d5	2	4

*Zuzüglich 1 Zyklus bei Seitenüberschreitung

```
Flags:  NVBDIZC
        *    **
```

CPX – Compare to register X

Die adressierten Daten werden vom X-Register abgezogen, das Ergebnis jedoch nicht gespeichert. Die Flags N,Z und C werden entsprechend gesetzt. Z = 1, wenn beide Werte gleich sind. N = 1 und C = 0, wenn X kleiner als die adressierten Daten ist. C = 1, wenn der Inhalt des X-Register größer oder gleich ist.

Adr.Art	Code	Länge	Zyklen
Abs:	$ec	3	4
OPge	$e4	2	3
imm	$e0	2	2

```
Flags:  NVBDIZC
        *    **
```

Laden aus dem Speicher

LDA – Load accumulator

Der Akku wird mit einem neuen Wert geladen.

Adr.Art	Code	Länge	Zyklen
Abs:	$ad	3	4
OPge	$a5	2	3
imm	$a9	2	2
Abs.X	$bd	3	4*
Abs.Y	$b9	3	4*
(Ind.X)	$a1	2	6
(Ind),Y	$b1	2	5*
OPge,X	$b5	2	4

*Zuzüglich 1 Zyklus bei Seitenüberschreitung

```
Flags:  NVBDIZC
        *    *
```

LDX – Load register X

Register X wird mit einem neuen Wert geladen.

Adr.Art	Code	Länge	Zyklen
Abs:	$ae	3	4
OPge	$a6	2	3
imm	$a2	2	2
Abs.Y	$be	3	4*
OPge,Y	$b6	2	4

*Zuzüglich 1 Zyklus bei Seitenüberschreitung

```
Flags:  NVBDIZC
        *    *
```

LDY – Load register Y

Register Y wird mit einem neuen Wert geladen.

Adr.Art	Code	Länge	Zyklen
Abs:	$ac	3	4
OPge	$a4	2	3
imm	$a0	2	2
Abs.X	$bc	3	4*
OPge,X	$b4	2	4

*Zuzüglich 1 Zyklus bei Seitenüberschreitung

```
Flags:  NVBDIZC
        *    *
```

Rückkehr aus Unterprogrammen

RTS – Return from subroutine

Der PC wird vom Stapel zurückgeholt und auf den nächsten Befehl gesetzt.

Adr.Art	Code	Länge	Zyklen
Implizit	$60	1	6

Flags: keine Veränderung

RTI – Return from interrupt

Stellt den ursprünglichen Zustand des Statusregisters und des PCs nach einer Programmunterbrechung wieder her.

Adr.Art	Code	Länge	Zyklen
Implizit	$40	1	6

```
Flags:  NVBDIZC
        *******
```

Sonstiges

BRK – break

Der PC und das Statusregister werden auf den Stapel gebracht. Als neue Adresse wird der Inhalt der Speicherstellen $fffe/$ffff übernommen. Zusätzlich wird das B-Flag gesetzt.

Adr.Art	Code	Länge	Zyklen
Implizit	$00	1	7

```
Flags:  NVBDIZC
          1 1
```

NOP – No operation

Wartet zwei Taktzyklen.

Adr.Art	Code	Länge	Zyklen
Implizit	$ea	1	2

Flags: keine Veränderung

Verschieben innerhalb der CPU

TAX – Transfer accumulator into register X.

Kopiert den Inhalt des Akkumulators ins X-Register

Adr. Art	Code	Länge	Zyklen
Implizit	$aa	1	2

```
Flags:  NVBDIZC
        *    *
```

TXA – Transfer register X into accumulator.

Kopiert den Inhalt des X-Registers in den Akkumulator.

Adr.Art	Code	Länge	Zyklen
Implizit	$8a	1	2

```
Flags:  NVBDIZC
        *    *
```

TAY – Transfer accumulator into register Y.

Kopiert den Inhalt des Akkumulators ins Y-Register

Adr. Art	Code	Länge	Zyklen
Implizit	$a8	1	2

```
Flags:  NVBDIZC
        *    *
```

TYA – Transfer register Y into accumulator.

Kopiert den Inhalt des Y-Registers in den Akkumulator.

Adr. Art	Code	Länge	Zyklen
Implizit	$98	1	2

```
Flags:  NVBDIZC
        *    *
```

TXS – Transfer register X into Stackpointer.

Kopiert den Inhalt des X-Registers in den Stapelzeiger.

Adr.Art	Code	Länge	Zyklen
Implizit	$9a	1	2

Flags: keine Veränderung

TSX – Transfer Stackpointer into register X.

Kopiert den Inhalt des Stapelzeigers ins X-Register

Adr.Art	Code	Länge	Zyklen
Implizit	$ba	1	2

```
Flags:  NVBDIZC
        *    *
```

Logische Operationen

AND – AND Accu

UND-Verknüpfung eines Arguments mit dem Akku. Das Ergebnis steht im Akku.

0 AND 0 = 0
1 AND 0 = 0
0 AND 1 = 0
1 AND 1 = 1

Adr.Art	Code	Länge	Zyklen
Abs:	$2d	3	4
OPge	$25	2	3
imm	$29	2	2
Abs.X	$3d	3	4*
Abs.Y	$39	3	4*
(Ind.X)	$21	2	6
(Ind),Y	$31	2	5*
OPge,X	$35	2	4

*Zuzüglich 1 Zyklus bei Seitenüberschreitung

```
Flags:  NVBDIZC
        *    *
```

ORA – Inclusive OR with accumulator

ORA-Verknüpfung eines Arguments mit dem Akku. Das Ergebnis steht im Akku.

0 ORA 0 = 0
1 ORA 0 = 1
0 ORA 1 = 1
1 ORA 1 = 1

Adr.Art	Code	Länge	Zyklen
Abs:	$0d	3	4
OPge	$05	2	3
imm	$09	2	2
Abs.X	$1d	3	4*
Abs.Y	$19	3	4*
(Ind.X)	$01	2	6
(Ind),Y	$11	2	5*
OPge,X	$15	2	4

*Zuzüglich 1 Zyklus bei Seitenüberschreitung

```
Flags:  NVBDIZC
        *    *
```

EOR – Exclusive-OR

EXKLUSIV-ODER Verknüpfung eines Arguments mit dem Akku. Das Ergebnis steht im Akku.

0 EOR 0 = 0
1 EOR 0 = 1
0 EOR 1 = 1
1 EOR 1 = 0

Adr.Art	Code	Länge	Zyklen
Abs:	$4d	3	4
OPge	$45	2	3
imm	$49	2	2
Abs.X	$5d	3	4*
Abs.Y	$59	3	4*
(Ind.X)	$41	2	6
(Ind),Y	$51	2	5*
OPge,X	$55	2	4

*Zuzüglich 1 Zyklus bei Seitenüberschreitung

```
Flags:  NVBDIZC
        *    *
```

BIT – Test bits

Die adressierten Bytes werden UND-verknüpft, das Ergebnis wird jedoch nicht festgehalten. Die Bits 6 und 7 der adressierten Speicherstelle werden in die Flags V und N übernommen. Der Akku bleibt unverändert.

Adr.Art	Code	Länge	Zyklen
Abs:	$2c	3	4
OPge	$24	2	3

```
Flags:  NVBDIZC
        mm   *
```

Stack-Manipulationen

PLA – Pull accumulator

Der Akku wird mit dem Inhalt der Stapelspitze geladen.

Adr.Art	Code	Länge	Zyklen
Implizit	$68	1	4

```
Flags:  NVBDIZC
        *    *
```

PHA – Push accumulator

Der Inhalt des Akku wird auf den Stapel gebracht.

Adr.Art	Code	Länge	Zyklen
Implizit	$48	1	3

Flags: keine Veränderung

PHP – Push processor status

Der Inhalt des Statusregisters wird auf den Stapel gebracht.

Adr.Art	Code	Länge	Zyklen
Implizit	$08	1	3

Flags: keine Veränderung

PLP – Pull processor status

Das Statusregister wird mit dem Inhalt der Stapelspitze geladen.

Adr.Art	Code	Länge	Zyklen
Implizit	$28	1	4

Flags: NVBDIZC

BVC – Branch if overflow clear

Testet das Überlaufsflag. Ist V = 0, wird zur nächsten Adresse plus dem angegebenen Abstand (Bereich von -128 bis +127) verzweigt. Bei gesetztem Flag erfolgt keine Aktion.

Adr.Art	Code	Länge	Zyklen
Relativ	$50	2	2*

* +1, bei Verzweigung
+2, bei Page-Überschreitung

Flags: keine Veränderung

BVS – Branch if overflow set

Testet das Überlaufsflag. Ist V = 1, wird zur nächsten Adresse plus dem angegebenen Abstand (Bereich von -128 bis +127) verzweigt. Bei gelöschtem Flag erfolgt keine Aktion.

Adr.Art	Code	Länge	Zyklen
Relativ	$70	2	2*

* +1, bei Verzweigung
+2, bei Page-Überschreitung

Flags: keine Veränderung

BCC – Branch if carry clear

Testet das Übertragsflag. Ist C = 0, wird zur nächsten Adresse plus dem angegebenen Abstand (Bereich von -128 bis +127) verzweigt. Bei gesetzem Flag erfolgt keine Aktion.

Adr.Art	Code	Länge	Zyklen
Relativ	$90	2	2*

* +1, bei Verzweigung
+2, bei Page-Überschreitung

Flags: keine Veränderung

BCS – Branch if carry set

Testet das Übertragsflag. Ist C = 1, wird zur nächsten Adresse plus dem angegebenen Abstand (Bereich von -128 bis +127) verzweigt. Bei gelöschtem Flag erfolgt keine Aktion.

Adr.Art	Code	Länge	Zyklen
Relativ	$b0	2	2*

* +1, bei Verzweigung
+2, bei Page-Überschreitung

Flags: keine Veränderung

Leider lassen sich beim 6510 die x- und y-Register nicht untereinander austauschen. Hier müssen Sie als Zwischenspeicher den Akku verwenden.

Zwei andere, weitaus gefährlichere Transportbefehle sind: TSX (Transfer Stackpointer into register X - kopiere den Inhalt des Stapelzeigers ins x-Register) und TXS (Transfer register X into Stackpointer - kopiere Stapelzeiger in x-Register). Der Sinn dieser Befehle ist zunächst nicht ganz klar. Aber bedenken Sie, daß es sonst keine Möglichkeit gibt, an den Inhalt des Stapelzeigers zu kommen. Ein anderer Anwendungszweck ist:

Nehmen wir an, Sie haben eine komplizierte Tastaturauswertung. Sie springen mit JSR von dieser in ein Unterprogramm, dieses ruft ein weiteres Unterprogramm auf (z.B. Speicherung auf Diskette). Was tun Sie, wenn bei der letzten Routine ein Fehler auftritt und Sie sofort zur Fehleranzeige springen wollen und von dort unmittelbar zurück zur Tastaturauswertung? Die JSR-Sprünge zwingen Sie, jeweils mit RTS die alte Reihenfolge zurückzuspringen. Wenn Sie sich allerdings vor dem ersten Unterprogrammaufruf den Stapelzeiger gemerkt haben (TSX), können Sie an einer x-beliebigen Position den Mikroprozessor mit TXS (+3) zwingen, zur ersten Position zurückzuspringen.

Stack-Manipulationen – PLA, PHA, PHP und PLP

Der 6510 hat leider nur drei programmierbare Register: x- und y-Register und den Akku. Für einige Anwendungen reicht die Anzahl der Register nicht aus, z.B für eine verzögerte Ausgabe einer Tabelle. Nehmen wir an, die Tabelle liegt bei $5000:

```
3A00 LDX #00
3A02 LDA 5000,X
3A05 JSR 3A0F
3A08 JSR FFD2
3A0B DEX
3A0C BNE 3A02
3A0E BRK

3A0F PHA
3A10 TXA
3A11 PHA
3A12 TYA
3A13 PHA
3A14 LDX #00
3A16 LDY #00
3A18 DEY
3A19 BNE 3A18
A1B DEX
3A1C BNE 3A18
3A1E PLA
3A1F TAY
3A20 PLA
3A21 TAX
3A22 PLA
3A23 RTS
3A24 F
```

In Zeile 3A00 erhält das x-Register den Wert Null. In der nächsten Zeile laden wir den ersten Wert unserer Tabelle. Danach springen wir ins Unterprogramm ab 3A0F (eine Verzögerungsschleife). Zurückgekehrt wird das Zeichen im Akku ausgegeben, das x-Register erniedrigt und die Schleife solange durchlaufen, bis das x-Register gleich null ist.

So weit, so gut. Wenn von Zeile 3A0F bis 3A13 nicht die Register gerettet, und ab 3A1E wieder zurückholen würden, käme das x-Register immer mit dem Wert Null aus dem Unterprogramm - unsere Routine hätte kein Ende. Der Befehl PHA (PusH accumulator - rette Akkuinhalt) bringt den Inhalt des Akkus auf den Prozessorstapel und erhöht den Stapelzeiger. Leider gibt es keinen Befehl für x- oder y-Register, darum muß zuerst der Wert der Register in den Akku gebracht werden, danach kann dieser auf den Stapel gerettet werden. Ab Zeile 3A1E geschieht das Umgekehrte: Mit PLA (PuLl Accumulator) wird der erste Wert wieder vom Stapel geholt und in den Akku übertragen. Wie wir vom Stapelprinzip her wissen, ist dies der zuletzt abgelegte Wert, also der des y-Registers. Er wird (Zeile 3A1F) auch wieder ins y-Register übertragen. Danach geschieht das gleiche auch mit dem x-Register und zum Schluß mit dem Akku.

Bei unserer kleinen Routine wäre es nur nötig gewesen, den Akku und das x-Register zu retten, doch ein bißchen Sicherheit ist besser. Denn falls diese Verzögerung von einer anderen Programmstelle aufgerufen wird, wissen wir nicht, welche Register wir vielleicht benötigen. Man könnte sogar noch mehr tun:

PHP (PusH Processor status) bringt das Statusregister auf den Stapel, und PLP (PuLl Processor status) bringt ihn wieder zurück vom Stapel ins Statusregister.

Interrupt – CLI, SEI, RTI

Mit drei Befehlen, die für die Interrupt-Behandlung zuständig sind, kommen wir zum Schluß unseres Kurses. Wie Sie bereits wissen, führt der C64 jede 60stel Sekunde einen Interrupt durch. Dieser Interrupt wird von einem der Timer-Bausteine ausgelöst. Es gibt außer den Timern noch viele Möglichkeiten, den IRQ auszulösen. Eines aber haben alle diese Routinen gemeinsam: Sie enden mit dem Befehl RTI (ReTurn from Interrupt - kehre vom Interrupt zurück). Im Gegensatz zu RTS muß RTI etwas mehr erledigen: RTS merkt sich, von welcher Position die Routine aufgerufen wurde, bei RTI kommt noch der Status dazu.

Trotzdem ist es manchmal interessant, den Interrupt auszuschalten: z.B. wenn die Speicherkonfiguration geändert wird. In Assembler haben wir die Möglichkeit, die kompletten 64 KByte Speicher als RAM zu adressieren. Dann geschieht beim Interrupt etwas Unangenehmes: Der Mikroprozessor versucht, in seinem Betriebssystem die IRQ-Routine aufzurufen. Wir haben allerdings auf RAM umgeschaltet. Daher findet der Prozessor sein Programm nicht - und hängt sich auf. Wir vermeiden dies mit SEI (SEt Interrupt mask - setze die Interrupt-Flagge). Im Status-Byte ist nämlich eines der Bits für den IRQ zuständig. Ist dieses Bit gesetzt, kann der Mikroprozessor keinen IRQ mehr ausführen. Ist es gelöscht, führt die CPU die Unterbrechungen wie gewohnt durch. Der Befehl zum Wiedereinschalten des IRQ ist CLI (CLear Interrupt flag).

Ein Nachzügler – NOP

Fast hätten wir ihn vergessen, den Befehl NOP (No OPeration - keine Tätigkeit). Er macht das, was sein Name sagt - nämlich zwei Taktzyklen lang nichts. Gebraucht wird er nur, falls man eine kurze Verzögerung in zeitkritischen Routinen benötigt und als Platzhalter bei geänderten Programmen.

Falls Ihnen dieser Kurs Appetit auf mehr Informationen gemacht hat, finden Sie im Anschluß noch einige wichtige Tabellen. Außerdem empfehlen wir unser Assembler-Sonderheft 35, zu bestellen bei Markt & Technik Leserservice, CSJ, Postfach 1 40 20, 8000 München 5, Tel. 089/20 25 15 28. Es beinhaltet einen noch ausführlicheren Kurs, bei dem auch Teile des Betriebssystems behandelt werden. Einige der dort beschriebenen Routinen finden Sie auch auf unserer Diskette. Als erste Programmierhilfe finden Sie rechts eine Tabelle zur Spritebehandlung. (Heimo Ponath, gr)



VIC - Video-Interface

Basis-Adresse = 53248 ($D000)

Sprites

... bauen sich aus einem Bit-Muster auf. Dabei werden 63 Byte benötigt. Pro Sprite stehen jeweils drei Byte in einer Bildschirmzeile. Aus 21 Zeilen besteht ein ganzes Sprite (Entwurf-Blatt). Bei Multicolor bilden je zwei Bit einen Zeiger auf eine der vier möglichen Farben:

00 = Hintergrund	01 = Farbe 1	10 = individuell	11 = Farbe 2

Sprite-Zeiger:

stehen im unsichtbaren Teil des Bild-Schirm-RAM. Sie zeigen auf den eigentlichen Sprite-Block. In ihm ist das Muster festgelegt. Die Zeiger beginnen ab Bildschirm + $03F8 im Text-Modus, also im Normalfall bei $07F8. Acht Speicherstellen sind für das jeweilige Sprite verantwortlich. Die Sprite-Blocks sind jeweils 64 Byte ($40) lang. Sie sind von $00 bis $FF **numeriert.** Berechnet wird ihre Position im Speicher folgendermaßen (VIC ist die derzeitig adressierte Offset-Adresse= $00 bei RESET):

VIC + (Blocknummer * 64) = Position

(nach RESET): 0 + (13 * 64) = 832

Koordinaten:

erscheinen in Reihenfolge x-Achse, y-Achse in den Registern. Bei Überschreitung des Offsets sind sie ganz oder teilweise vom Bildschirmrand verdeckt.

Offset für x-Koordinate: < 24
Offset für y-Koordinate: < 29 oder > 229

$D000	**x-Koordinate Sprite #0**	$D008	**x-Koordinate Sprite #4**
$D001	**y-Koordinate Sprite #0**	$D009	**y-Koordinate Sprite #4**
$D002	**x-Koordinate Sprite #1**	$D00A	**x-Koordinate Sprite #5**
$D003	**y-Koordinate Sprite #1**	$D00B	**y-Koordinate Sprite #5**
$D004	**x-Koordinate Sprite #2**	$D00C	**x-Koordinate Sprite #6**
$D005	**y-Koordinate Sprite #2**	$D00D	**y-Koordinate Sprite #6**
$D006	**x-Koordinate Sprite #3**	$D00E	**x-Koordinate Sprite #7**
$D007	**y-Koordinate Sprite #3**	$D00F	**y-Koordinate Sprite #7**

$D010 neuntes Bit der x-Koordinaten

Ein gesetztes Bit bedeutet, daß die x-Koordinate >255 ist.

Bit	7	6	5	4	3	2	1	0
Sprite	7	6	5	4	3	2	1	0

Farb-Register:

legen im Normalmodus die Farbe eines Sprites fest (0 bis 16). Bei Multicolor wird nur die die individuelle Farbe (%10) bestimmt.

$D027	**Sprite #0**	$D02B	**Sprite #4**
$D028	**Sprite #1**	$D02C	**Sprite #5**
$D029	**Sprite #2**	$D02D	**Sprite #6**
$D02A	**Sprite #3**	$D02E	**Sprite #7**

$D01C Multicolor-Modus

Ein gesetztes Bit schaltet das entspr. Sprite in Mulitcolor.

Bit	7	6	5	4	3	2	1	0
Sprite	7	6	5	4	3	2	1	0

$D025 und $D026 Farben für Multicolor

Bei Multicolor sind für Sprites jeweils zwei Farben identisch, Farbe 1 ($D025) und Farbe 2 ($D026).

$D010 Sprites ein/aus

Ein gesetztes Bit schaltet das entsprechende Sprite ein.

Bit	7	6	5	4	3	2	1	0
Sprite	7	6	5	4	3	2	1	0

$D01B Priorität Sprite/Hintergrund

Ein gesetztes Bit schiebt das entspr. Sprite hinter den Text.

Bit	7	6	5	4	3	2	1	0
Sprite	7	6	5	4	3	2	1	0

Entwurf-Blatt:

Byte 1	Byte 2	Byte 3

$D017 vertikale Vergrößerung

Ein gesetztes Bit vergrößert das entspr. Sprite nach rechts

Bit	7	6	5	4	3	2	1	0
Sprite	7	6	5	4	3	2	1	0

$D01D horizontale Vergrößerung

Ein gesetztes Bit vergrößert das entspr. Sprite nach unten.

Bit	7	6	5	4	3	2	1	0
Sprite	7	6	5	4	3	2	1	0

$D01E Kollision Sprite/Sprite

Ein gesetztes Bit zeigt welche Sprites beteiligt waren.

Bit	7	6	5	4	3	2	1	0
Sprite	7	6	5	4	3	2	1	0

$D01F Kollision Sprite/Hintergrund

Ein gesetztes Bit zeigt welche Sprites beteiligt waren.

Bit	7	6	5	4	3	2	1	0
Sprite	7	6	5	4	3	2	1	0

$D01A Interrupt Maskenregister

legt fest welches Ereignis einen IRQ auslöst

Bit	
0	Raster-Interrupt (1 = ja)
1	Sprite/Hintergrund Kollision (1 = ja)
2	Sprite/Sprite Kollision (1 = ja)
3	Lightpen (1 = ja)
4-7	Unbelegt (immer 1)

$D019 Interrupt Statusregister

zeigt, welches Ereignis einen IRQ ausgelöst hat.

Bit	
0	Raster-Interrupt (1 = ja)
1	Sprite/Hintergrund Kollision (1 = ja)
2	Sprite/Sprite Kollision (1 = ja)
3	Lightpen (1 = ja)
4-6	Unbelegt (immer 1)
7	Interrupt fand statt (1 = ja)

Stichwort Monitor: **Doch diesmal ist nicht das Gerät gemeint, das neben dem Computer steht, sondern Software zur Behandlung von Maschinensprache; mit Diskettenmonitor und einem Disassembler, der sogar illegale Opcodes erkennt.**

SMON - der Maschinensprache-Monitor

Wenn der

Jedem Programmierfreak ist es schon passiert, daß sein Werk aus zunächst unerfindlichen Gründen abgestürzt ist. Das Überprüfen des Programmablaufs nach Schreibfehlern ist äußerst zeitintensiv und führt nicht immer zum Erfolg. Spätestens jetzt wird ein guter Monitor nötig. Erst mit ihm wird es ohne großem Aufwand möglich, ein Programm durchzutesten. Dazu existieren über 35 Befehle. Wem das nicht genügt, für den gibt es als Zusatz die Erweiterungen »Disk-SMON«, »SMON-Extra« und »SMON-Illegal«.

Im Gegensatz zu den schon vorgestellten Hypra-Ass und Hypra-Reass, ist der SMON auf spontanes Arbeiten ausgerichtet, d.h. er verwendet keine abspeicherbaren Quelltexte, sondern direkte Eingaben. Auf der beiliegenden Diskette befinden sich mehrere Grundversionen des SMON. Es sind dies:

1. »SMON $C000« - für den C64 alt.
2. »SMON $C000 II« - für den C64 neu.
3. »SMON 128D« - für den C128 im 64'er-Modus.

Alle drei Versionen belegen den Speicher ab 49152 ($C000 bis $CFFF). Welche Version die richtige ist, merken Sie bei Verwendung des Trace-Befehls (s.d.). Geladen wird unmittelbar. Beispielsweise mit:

```
LOAD"SMON $C000",8,1
```

Danach ist die Eingabe von NEW wichtig, damit alle Basic-Pointer wieder geradegerückt werden. Im SMON ergeben sich zwar außer beim Floppymonitor keine Schwierigkeiten, da alle Routinen von Haus aus auf richtige Werte gesetzt werden. Ohne NEW erhalten Sie allerdings spätestens nach Verlassen des SMON bei Lade- oder Speicheroperationen einen »OUT OF MEMORY ERROR?«.

Gestartet wird diese Version mit

```
SYS49152
```

Danach meldet sich der SMON mit einer Veränderung der Bildschirmfarben und der Anzeige der Prozessorregister:

```
 PC  SR AC XR YR SP  NV-BDIZC
;C00B B0 C2 00 00 F2  10110000
.■
```

Neben einem Punkt (Prompt) erscheint der Cursor. Er läßt sich wie gewöhnt mit den Cursortasten bewegen. In der ersten Zeile befinden sich die Registeranzeigen (s.a. »R«).

Vor der Erklärung der Befehle eine kurze Beschreibung der im Artikel verwendeten Synonyme:

Der Monitor benötigt für alle Eingaben von Zahlen die hexadezimale Schreibweise. Um dezimal auf hexadezimal und umgekehrt umzurechnen, besitzt er die entsprechenden Funktionen.

Ein Befehl bzw. eine Befehlsfolge werden neben dem Prompt (».«, »,«, »;« usw.) eingetippt und mit <RETURN> bestätigt.

»Start« - eine 4stellige Hexadezimalzahl (0000 bis FFFF), die den Beginn der Operation einleitet.

»Ende« - ebenfalls eine 4stellige Hexadezimalzahl - gibt das Ende der Operation an.

»Bytes« - sind eines oder mehrere Bytes, mit denen die Funktion ausgeführt wird. Sie werden 2stellig eingegeben (00 bis FF). Mehrere Bytes werden durch »Spaces« (Leerzeichen) getrennt.

Die Bildschirmausgabe kann zusätzlich (bis auf Trace) auf den Drucker umgeleitet werden, wenn Sie die Befehle »geshiftet« (d.h. zusammen mit <SHIFT> eingeben).

Wird bei einem Befehl, bei dem Start und Ende verlangt ist, nur »Start« eingegeben, liefert der SMON zunächst eine Zeile und wartet dann auf einen Tastendruck. <SPACE> tastet zeilenweise und <RUN/STOP> bricht die Ausgabe ab. Jede andere Taste führt zu einer fortlaufenden Anzeige. Sie wird wieder mit <RUN/STOP> abgebrochen.

1. Einfache Monitorbefehle

X - Verlassen des Monitors

... alle Basic-Pointer werden wie vor Aufruf von SMON rekonstruiert und der SMON nach Basic verlassen.

R - Anzeige der Register

... zeigt die aktuellen Registerinhalte an. Sie können durch Überschreiben geändert werden.

PC	SR	AC	XR	YR	SP
Programmzähler	Statusregister	Akkumulator	X-Register	Y-Register	Stapelzeiger

Das Statusregister wird bitweise dargestellt (NV-BDIZC):

- N= Negative-Flag
- V= Overflow-Flag
- -= Unbelegt
- B= Break-Flag
- D= Dezimal-Flag
- I = Interrupt-Dissable-Flag
- Z= Zero-Flag
- C= Carry-Flag

M Start Ende - Anzeige der Speicherinhalte

... erlaubt Anzeigen und Ändern des Speicherinhalts im Bereich von Start bis Ende. Die Anzeige erfolgt zu jeweils acht Byte in einer Zeile mit deren ASCII-Codes im Anschluß. Beispiel:

```
.M20002200
```

zeigt den Speicherbereich von $2000 bis $2200. Nach <RETURN> füllt die Anzeige zuerst den Bildschirm, dann scrollt er durch. Mit <CONTROL> kann die Anzeige verlangsamt werden. <RUN/STOP> bricht die Ausgabe ab.

Wählen Sie für die Bildschirmdarstellung einen geringeren Bereich (z.B. »M200020A0«), oder tasten Sie sich durch den Bereich (»M2000«, danach <SPACE>). Für die Druckerausgabe sind allerdings die gesamten Bereiche interessanter (»m20002200«).

O Start Ende Byte - Füllen von Speicherbereichen

... füllt den Speicher im Bereich von Start bis Ende mit dem Wert von Byte:

```
O 8000 9FFF 00
```

G Adresse - Sprung in ein Maschinenprogramm

... startet ein Maschinenprogramm ab »Adresse«. Adresse muß 4stellig hexadezimal eingegeben werden (z.B. »G C000«). Es ist sinnvoll, die Programme mit »BRK« (Break) abzuschließen, da beim BRK zurück in den Monitor gesprungen wird. »RTS« als Abschluß eines Programms führt zurück in den Basic-Interpreter.

Durchblick kommt

2. Eingabe und Ausgabeoperationen

PO Gerätenummer - Einstellung der Geräteadresse für den Drucker

... setzt die Primäradresse für den Drucker auf »Gerätenummer«. Voreingestellt ist »4« (für Commodore-Printer):

PO 6

ändert die Druckeradresse auf »6« (Plotter).

IO Gerätenummer - Einstellung der Geräteadresse für LOAD und SAVE

... stellt die Standard-Gerätenummer für alle LOAD- und SAVE-Operationen auf den mit Gerätenummer bezeichneten Wert. Voreingestellt ist »8« (Floppy).

S"Name"Start Ende - Speichern

... überträgt den Bereich von Start bis Ende auf das durch »IO« voreingestellte Gerät. Beachten Sie dabei, daß Ende auf das Byte nach dem Programm stehen muß. So speichert:

S"SMON"C000 D000

den Bereich von $C000 bis einschließlich $CFFF.

L"Name"Start - Laden

... lädt ein Programm von dem Gerät, das mit »IO« bestimmt wurde. Start ist optional und ermöglicht ein Laden an eine andere Speicherposition als die, für die das Programm geschrieben ist. Beispiel:

L"SMON $C000"2000

lädt den SMON ab der Speicherstelle $2000. Lassen Sie die Adressenangabe (2000) fort, wird das Programm an die Originaladresse geladen ($C000).

3. Umrechnungsfunktionen

#Dezimalzahl - Umrechnung Dezimal in Hexadezimal

... rechnet »Dezimalzahl« in Hexadezimal um. Falls der Wert kleiner als 256 ist, wird zusätzlich in Binär umgewandelt:

.#123

ergibt »7B 01111011 123«. Wogegen bei

#1024

»0400 1024« am Bildschirm erscheint

$Hexadezimalzahl - Umrechnung Hexadezimal in Dezimal

... rechnet »Hexadezimalzahl« in Dezimal um. Falls der Wert kleiner als $0100 ist, wird zusätzlich in Binär umgewandelt.

%(Binärzahl) - Umrechnung Binär in Hexadezimal und Dezimal

... wandelt eine achtstellige Binärzahl (z.B. %00000011) in den entsprechenden Hexadezimal- und Dezimalwert.

? - Rechnen im SMON

... erlaubt Addition oder Subtraktion mit zwei Zahlen. Die Zahlen müssen in 4stellig hexadezimal eingegeben werden. Beispielsweise »?0023+0023« ergibt »46 01000110 70«.

4. Vergleichs- und Suchfunktionen

= Start1 Start2 - Vergleichen eines Speicherbereichs

Vergleicht den Speicherbereich ab Start1 mit dem Bereich ab Start2. Die erste abweichende Speicherstelle wird angezeigt und der Vergleich abgebrochen:

= C000 CA00

Vergleicht den Speicherbereich ab $C000 mit dem ab $CA00, daraus entsteht:

=.C000 CA00 C000

Beispiel über das Verschieben eines Programms

Da Üben besser ist als alle Theorie, passen wir jetzt unseren SMON an auf die Speicherposition $8000 bis $8FFF. Verschieben Sie ihn zuerst wie unter »W« beschrieben nach $8000. Dann rechnen Sie mit

V C000 CFFF 8000 820B 8FD2

den Bereich $820B bis $8FD2 auf die neuen Sprungadressen ($8000 bis $8FFF) um. Dabei werden als Suchreferenz die Adressen verwendet, die $C000 bis $CFFF verwendet hatten. Wir lassen $8000 bis $820A und ab $8FD3 aufwärts beim Umrechnen aus, da sich hier Tabellen befinden.

Achtung: Es lassen sich grundsätzlich nur 3-Byte-Befehle umrechnen, Sprungtabellen und Vektoren müssen Sie per Hand ändern:

M 802B 806B

bringt Ihnen eine Auflistung der Sprungtabellen. Wir kennen diesen Bereich, bei einem fremden Programm müßten Sie ihn erst suchen.

Als erste Ausgabezeile erhalten Sie:

:802B DA CA 2D C9 07 C9 1B C9 (+ ASCII-Zeichen)

Bei diesen Sprungtabellen beziehen sich jeweils zwei Bytes (low-, high-Byte) auf ein Sprungziel. Beispielsweise bedeutet »DA CA« einen Sprung auf die Position $CADA. Wir benötigen aber einen Sprung in unser geändertes Programm (nach $8ADA). Daher ändern Sie in dieser und den darauffolgenden Zeilen das erste Zeichen jedes zweiten Bytes:

:802B DA 8A 2D 89 07 89 1B 89 (hier nichts ändern)

Wenn Sie das letzte Byte einer Zeile geändert haben (C9 in 89), bestätigen Sie mit <RETURN>, erst damit sind die Änderungen übernommen.

Auch die Adressen der Immediate-Adressierung müssen von Hand geändert werden. Für sie gibt es den Suchbefehl »FI«:

.FIC*,8000 8FFA

Sie erhalten eine Reihe von disassemblierten Befehlen, die Sie wie folgt behandeln:

```
,8005 A9 C2  LDA #C2 -- ändern in LDA #82
,8124 E0 C0  CPX #C0 -- nicht ändern
,8386 A0 C0  LDY #C0 -- ändern in LDY #80
,8441 C9 C0  CMP #C0 -- nicht ändern
,887F A2 C3  LDX #C3 -- nicht ändern
,888D A2 C1  LDX #C1 -- nicht ändern
,8992 A9 C1  LDA #C1 -- nicht ändern
,8C2C A9 CC  LDA #CC -- ändern in LDA #8C
,8C5B A9 C2  LDA #C2 -- ändern in LDA #82
,8CF4 A9 CC  LDA #CC -- ändern in LDA #8C
,8DA1 A2 CC  LDX #C2 -- ändern in LDX #82
,8E03 A9 CC  LDA #CC -- ändern in LDA #8C
,E6C5 A9 C0  LDA #C0 -- nicht ändern
,8F71 A0 CF  LDY #CF -- ändern in LDY #8F
```

Sie sehen, es gibt keine Regel, welche Befehle zu ändern sind und welche nicht. Aus diesem Grund sind diese Änderungen von Hand vorzunehmen. Bei fremden Programmen müssen Sie den Befehlsfolgen nachgehen und analysieren, wo Sprungadressen geladen oder manipuliert werden.

Zuletzt gibt es noch eine Sprungtabelle im Diskettenmonitor, die geändert werden muß:

M 8FD8 8FE4

Bei den jetzt sichtbaren zwei Bildschirmzeilen ändern Sie, wie bei der ersten Tabelle beschrieben, das »C« jedes zweiten Bytes in »8« (ab »55« in der zweiten Zeile ist Schluß).

Jetzt müssen Sie nur noch Ihr Werk abspeichern:

S"SMON $8000",8000 8FFF

F Bytes, Start Ende - Suche nach Bytefolgen

... sucht im Speicher des C 64 ab Adresse (Start) bis (Ende) nach der Bytefolge (Bytes). Sind die Bytes gefunden, so erscheint die Speicheradresse auf dem Bildschirm.

Achtung: Nach »F« muß ein Leerzeichen folgen, mehrere Bytes werden jeweils durch ein Leerzeichen getrennt. Die Eingabe darf zwei Bildschirmzeilen nicht überschreiten (Leerzeichen zwischen Start und Ende dürfen weggelassen werden). Ohne die Angabe von Start und Ende untersucht SMON den gesamten Speicher nach dem(n) Byte(s). Beispiel:

```
.F 00 FF,2000 3000
```

FAAdresse,Start Ende - Suche nach absoluten Adressen

... findet im Bereich Start bis Ende alle Befehle, die »Adresse« als absoluten Operanden haben.

Achtung: Zwischen »FA« und »Adresse« darf kein Leerzeichen sein:

```
FAC2C2,C000CFFF
```

... disassembliert alle Befehle, die mit dem Operanden $C2C2 versehen sind (in unserem Beispiel nur »JSR«).

Die Adresse muß nicht vollständig eingegeben werden, pro Stelle läßt sich das Jokerzeichen »*« einfügen. Als Beispiel sollen alle Befehle erscheinen, bei der SMON auf den Grafikbereich ($D000 bis $DFFF) zugreift:

```
FAD***,C000CFFF
```

FRAdresse,Start Ende - Suche nach relativen Adressen

... hier wird im Bereich Start bis Ende nach Branch-Befehle gesucht, die zu »Adresse« verzweigen. Diese Befehle lassen sich mit »FA« nicht finden, da nach dem Befehlscode keine absolute Adresse steht, sondern z.B. »verzweige um 10 vorwärts«, oder »verzweige um 100 rückwärts«. Beispiel: Gesucht werden alle Branch-Befehle, die zu $C280 verzweigen.

```
.FRC280,C000CFFF
```

Natürlich können solche Befehle nur maximal 128 Bytes vom Sprungziel entfernt sein. Die Bereichsangabe ist also in unserem Beispiel viel zu groß gewählt. (SMON stört dies allerdings nicht). Der Einsatz des Jokers ist auch hier, wie unter »FA« beschrieben, möglich.

FTStartEnde - Suche nach Tabellen

... durchforstet den Bereich Start bis Ende nach Tabellen. Dabei wird alles, was sich nicht disassemblieren läßt, als Tabelle behandelt und ausgegeben:

```
.FTC000CFFF
```

erzeugt eine schier endlose Folge von Adressen mit unbekannten Zeichen. Hier ist die Druckerausgabe sinnvoller. Anhand der Liste lassen sich anschließend die Speicherbereiche nach Tabellenbereichen überprüfen (s. »M« und »K«).

FZAdresse,StartEnde - Suche nach Befehle mit Zero-Page-Adressen

... überprüft den Bereich Start bis Ende nach Befehlen, die »Adresse« ($00 bis $FF) ansprechen:

```
FZ01,C000CFFF
```

... findet alle Befehle im SMON, bei denen die Zero-Page-Adresse $01 verwendet wird ($01 = Speicheraufteilung des C64).

FIOperand,StartEnde - Suche nach Befehlen mit unmittelbarer Adressierung (immediate)

... findet Befehle, deren Operanden immediate adressiert sind (z.B. LDA #02, LDX #02 usw.):

```
FI02,C000CFFF
```

da schon die Speicherstelle $C000 einen anderen Wert als die $CA00 hat.

5. Adreßumrechnungs- und Verschiebe-Befehle

W Startalt Endealt Startneu - Verschieben ohne Umrechnung

... schiebt den Inhalt der Speicherstellen von Startalt bis Endealt in den Bereich ab Startneu. Dabei handelt es sich nicht um ein echtes Verschieben, sondern um ein Kopieren. Lediglich beim Überlappen beider Bereiche, verändert sich der alte Inhalt. Ansonsten ist der neue Speicherinhalt mit dem alten identisch.

```
W C000 CFFF 8000
```

kopiert den SMON ($C000 bis $CFFF) nach $8000 bis $8FFF.

V Startalt Endealt Startneu Start Ende - Umrechnung ohne Verschieben

... rechnet alle Sprungadressen im Bereich Start bis Ende um. Es findet keine Verschiebung des Programms statt. Der Bereich Startalt bis Endealt wird als Suchkriterium für die alten Adressen genommen und für Startneu umgerechnet. Haben Sie z.B. ein Programm an eine andere als die vorgesehene Adresse geladen, hilft Ihnen diese Funktion, das Programm lauffähig zu machen.

C Startalt Endealt Startneu Start Ende - Verschieben mit Adreßumrechnung

... ist die Zusammenfassung von »W« und »V«. Startalt bis Endealt wird nach Startneu geschoben und alle Sprungadressen der 3-Byte-Befehle von Start bis Ende umgerechnet. Beachten Sie, daß Start und Ende für den neuen Bereich angegeben werden müssen. Dazu ein (theoretisches) Beispiel:

```
C 4000 4020 400B 4000 4011
```

verschiebt ein Programm von $4008 bis $4020 zur neuen Anfangsadresse $400B. Dabei werden im Bereich von $4000 bis $4011 die Sprungadressen umgerechnet. Jetzt könnten Sie beispielsweise ab $4008 einen 3-Byte-Befehl einfügen. Die Funktion »C« eignet sich für kleinere Programme. Bei größeren ist es sinnvoller, zuerst mit »W« zu verschieben und danach mit »V« umzurechnen.

6. Sonderfunktionen und Befehle

B Start Ende - Wandlung in Basic-Zeilen

... wandelt ein Maschinenprogramm von Start bis Ende-1 in Basic-Data-Zeilen.

```
.BC000 CFFF
```

wandelt den SMON in Basic-Datas. Nach der Wandlung befindet sich der C64 im Basic-Interpreter. Die Data-Zeilen lassen sich mit

```
SAVE"XXXXX",8
```

speichern. Der SMON muß anschließend neu gestartet werden.

K Start Ende - Ausgabe von ASCII-Zeichen

... listet die ASCII-Zeichen des Bereichs Start bis Ende. Es werden jeweils 32 Zeichen pro Zeile ausgegeben. Sie lassen sich per Tastatur überschreiben:

```
.K 0801
```

Zeigt eine Zeile (32 Zeichen) ASCII-Zeichen ab Basic-Speicherbeginn. Wie auch bei »M«, läßt sich mit <SPACE> weitertasten und mit <RETURN> eine fortlaufende Ausgabe erreichen.

7. Disassembler

D Start Ende - Ausgabe von Mnemonics

... disassembliert den Bereich Start bis Ende. Wird Ende nicht eingegeben, erscheint zunächst nur eine Zeile (<SPACE> = tasten, <RETURN> = fortlaufend, <RUN/STOP> = abbrechen). Die Befehle lassen sich überschreiben (z.B. »LDA #01« in »LDA #00«). Die vor den Mnemonics angezeigten Hex-Bytes ändern sich automatisch. Sie lassen sich allerdings nicht überschreiben. Illegale Codes (Befehle, die nicht den Mnemonics entsprechen) werden mit drei Sternchen ausgegeben.

8. Direkt-Assembler

A Start - Beginn des Assemblierens

... leitet das Assemblieren ab »Start« ein. Dabei lassen sich Label verwenden. Beim SMON beginnt ein Label mit »M«

(Marke), gefolgt von einer 2stelligen Hexadezimal-Zahl (00 bis 30). Es bezieht sich nur auf Sprungbefehle.

Verlassen wird der Assembler mit

F - Beenden des Assemblierens

Danach wird der komplette assemblierte Bereich noch einmal disassembliert angezeigt.

Beispiel für den Direkt-Assembler:

`.A 8000`

damit zeigen Sie an, daß Sie ab Speicherposition $8000 mit dem Assemblieren beginnen wollen. In der nächsten Bildschirmzeile erscheint:

`8000 ■`

Der Cursor (neben $8000) fordert Sie zur Eingabe von Mnemonics ein:

`8000 LDA #00`

Der Assembler macht daraus

`8000 A9 00   LDA #00`

`8002 ■`

als nächste Eingabe folgt:

```
8002 LDX #04
8004 STA FE
8006 STX FF
8008 M01 TYA
```

Nach der letzten Eingabe geschieht etwas Seltsames: »M01« verschwindet. Intern merkt sich SMON aber diese Marke (im Kassettenpuffer). Sie müssen bis jetzt am Bildschirm sehen:

```
8000 A9 00 LDA #00
8002 A2 04 LDX #04
8004 85 00 STA #00
8006 86 04 STX #04
8008 98 TYA
8009 ■
```

Führen Sie das Programm weiter fort mit:

```
8009 STA (FE),Y
800B DEY
800C BNE M01
800E BRK
800F F
```

Nach der Eingabe von »F« wird das komplette Programm nochmals ausgegeben und der Branch-Befehl bei $800C deutet jetzt auf $8008.

Achtung: Mit »F« wird der Eingabemodus verlassen. Erst danach startet SMON den Assemblierungs-Vorgang.

9. Trace-Modus

TWStart - Trace-Walk

... startet ein Maschinenprogramm ab Start im Einzelschritt-Modus. Dabei wird ein Befehl ausgeführt und der nächste angezeigt, danach wartet SMON auf einen Tastendruck. <SPACE> führt den nächsten Befehl durch, <RUN/STOP> unterbricht Trace-Walk und gibt die Register aus. Wenn Sie das Beispiel unter »A« eingetippt haben, starten Sie dieses kleine Utility mit:

`TW8000`

Als erste Anzeige erhalten Sie

`8002 23 00 04 E2 F6   LDX #04`

Dies ist korrekt, denn der erste Befehl ist schon ausgeführt und der zweite wird angezeigt. Neben »8002« erscheinen die Register in der Reihenfolge:

```
     SR AC XR YR SP
8002 23 00 04 E2 F6   LDX #04
```

(Erklärung s. »R«). Damit lassen sich Programme bequem überprüfen. Wenn Sie mit feststehenden Werten starten möchten, empfiehlt es sich mit »R« zuerst die Register aufzurufen, und zu überschreiben. Ein Beispiel: Ändern Sie den Wert unter »YR« (y-Register) durch Überschreiben auf »01«, bestätigen Sie diese Zeile mit <RETURN> und starten Sie Trace-Walk mit:

`TW8008`

Nachdem Sie vier Zeilen mit <SPACE> durchgetastet haben, stößt SMON auf »BRK« und beendet die Ausgabe mit der Registeranzeige. Wenn Sie sich jetzt die Zahl unter »YR« ansehen (»00«) und wie oben mit »TW8008« starten, benötigen Sie 255 x 4 Tastungen, um zum Ende der Schleife zu gelangen.

Achtung: Wenn Sie einen verkehrten SMON geladen haben, tastet Trace ständig den ersten Befehl. Probieren Sie dann eine der anderen Versionen. Beispielsweise mit:

`LOAD"SMON $C000 II",8,1`

und NEW. Danach starten Sie wieder mit SYS 49152. Mit der richtigen Version des SMON funktioniert Trace ohne Störungen.

TSStart Ende - Trace-Stop

... beginnt ein Maschinenprogramm ab Start und bricht bei Ende mit der Registeranzeige ab. Die Registeranzeigen lassen sich prüfen und ggf. überschreiben. Mit »G«, »TW« oder »TB« ohne weitere Adreßeingaben starten Sie erneut (mit Übernahme der neuen Register). Sinnvoll ist dieser Befehl, wenn längere Programme »getraced« werden sollen, der Anfang aber durchlaufen werden muß (z.B. bei Tastatureingaben).

Achtung: »TS« funktioniert nur im RAM des C64, da es einen Stop-Befehl ins Programm schreibt, und später wieder rekonstruiert. Aus dem gleichen Grund darf die Adresse, bei der gestoppt wird, vom Programm nicht modifiziert werden.

TSPosition Anzahl - Trace-Break

... setzt einen Breakpoint bei Speicherposition »Position« für eine »Anzahl« von Programmdurchläufen. Das heißt später wird mitgezählt, wie oft »Position« im Programm durchlaufen wird und der Programmablauf wird nach »Anzahl« abgebrochen. Nach der Eingabe dieses Befehls passiert zunächst nichts. »TS« benötigt zur Ausführung den Befehl:

TQStart - Trace-Quick

Er startet ein Programm ab »Start«. Ist die in »TS« festgelegte Anzahl der Programmdurchläufe erreicht, stoppt der Ablauf mit Anzeige der Register.

10. Disketten-Monitor

Im Gegensatz zu den anderen Befehlen des SMON ist sein Wortschatz nicht direkt erreichbar. Sie müssen den Diskmon erst mit ».Z« aufrufen.

Z - Aufruf des Diskmon

... ändert die Rahmenfarbe auf Gelb und springt in den Befehlsinterpreter des Disk-Mon. Vor dem Cursor erscheint zur Unterscheidung zum normalen SMON-Modus ein »*«. Von $BF00 bis $C000 (im RAM unterm ROM) wird ein Puffer zur internen Speicherung von Daten reserviert. In diesem Speicher lassen sich die Diskettendaten einlesen und können durch Überschreiben geändert werden. Danach lassen sie sich wieder zurückschreiben. Alle folgenden Befehle zeigen nur im Diskmon die beschriebene Wirkung.

X - Beenden des Diskmon

... verläßt den Diskmon und kehrt zum SMON zurück.

RTrackSektor - Lesen der Daten von Diskette

... liest »Track« und »Sektor« von der Diskette in den Speicher des C64. Läßt man die Parameter weg, werden nachfolgender Track und Sektor gelesen. Das heißt Track und Sektor, auf die der Blockverbinder der zuletzt eingelesenen Daten zeigt. Unmittelbar nach dem Einlesen erscheint die erste Zeile der Daten am Bildschirm. Lassen Sie sich nicht von der Bezeichnung »BFxx« am Anfang jeder Zeile irritieren. Der Diskmon verwendet die Ausgaberoutinen des SMON und gibt daher den Speicherplatz im C64 bekannt. Für Ihre Analysen interessieren nur die Stellen nach »BF«. Sie zeigen die Position auf dem gelesenen Sektor.

Achtung: Anders als beim SMON, gibt der Diskmon die weiteren Blockdaten durch <SHIFT> aus. Das geschieht so lange, wie Sie die Taste drücken, bzw. bis der komplette Block angezeigt ist.

WTrackSektor - Schreiben von Daten auf Diskette

... schreibt einen Datenblock zurück auf die Diskette. Ähnlich wie bei »R« kann die Angabe »Track/Sektor« entfallen. Er wird dann an die Position der zuletzt unter »R« gelesenen Daten geschrieben. Das ist (fast) immer der richtige.

M - Memory-Dump

... gibt die Blockdaten am Bildschirm aus. Die Tasten <SHIFT> und <RUN/STOP> sind dabei, wie unter »R« beschrieben, aktiv.

@ - Lesen des Fehlerkanals

... liest den Fehlerkanal der Floppy-Station, gibt ihn aber nur aus, wenn tatsächlich ein Fehler vorhanden war.

Mit dem SMON und seinen einzelnen Erweiterungen steht Ihnen ein tolles Werkzeug zur Analyse von Maschinenprogrammen zur Verfügung.

Erweiterungen des SMON

1. Disk-SMON

... bietet erweiterte Möglichkeiten im Diskettenmonitor. Dafür entfallen die Trace-Befehle des SMON. Laden Sie zuerst die richtige $C000-Version für Ihren C64, dann das Overlay-Programm mit

```
LOAD"FLOPPYMON",8,1
```

Nach der Eingabe von NEW starten Sie SMON mit »SYS49152«. Jetzt müssen Sie zuerst mit

```
G CDD8
```

die neuen Befehle initialisieren. Danach ist es empfehlenswert, die neue Version auf Ihre Arbeitsdiskette zu speichern:

```
S"FLOPPYSMON $C0"C000 CFFF
```

Ihr Disk-Mon ist jetzt einsatzbereit. Er wird aus dem Floppymon im SMON gestartet, d.h. Sie müssen zuerst ».Z« eingeben. Dadurch wird der interne Floppymon initialisiert. Danach führt

```
*F
```

in den erweiterten Diskettenmonitor. Verlassen (zum normalen Floppymon) wird er mit »X«. Es stehen drei neue Befehle zur Verfügung:

M - Memory-Dump

... listet das Floppy-RAM oder ROM mit seinen aktuellen Speicherwerten. Durch Überschreiben können diese Werte geändert werden, <RETURN> sendet die geänderte Zeile wieder zur Floppy. »M« ohne Zusätze zeigt die Zero-Page des Computers der Floppy an. Anders als in den anderen Modi des SMON wird die Anzeige mit <SPACE> weitergetastet. <RUN/STOP> bricht ab.

Achtung: Da diese Änderungen unterhalb der Betriebssystem-Ebene der Floppystation stattfinden, ist eine Zerstörung des Disketteninhalts sehr schnell passiert. Verwenden Sie niemals ein Original, sondern grundsätzlich eine Kopie der zu manipulierenden Diskette.

V Start Puffer - Verschieben

... schiebt 256 Byte eines Maschinenprogramms ab Start aus dem Speicher des C64 in den Floppy-Puffer »Puffer«.

```
V 6000 0400
```

... überträgt 256 Byte ab Speicherposition $6000 in den Floppypuffer »1« (0400 = Puffer 1, s.u.).

@ - Floppybefehle

... überträgt Befehle des DOS 5.1 zur Floppy. Beispiele:

»@« - ohne Zusätze liest den Fehlerkanal.

@I« - initialisiert die Diskette

»@S:name« - löscht das File »name« von Diskette (scratch).

Tips zum Umgang mit dem Disk-SMON

Die Diskettenstation besitzt einen eigenen Mikroprozessor und damit ein eigenes Betriebssystem. Anders als im C64 geschieht in der Floppystation die Befehlsausführung Interrupt-gesteuert. Das heißt in bestimmten Zeitabständen prüft der Mikroprozessor der Floppy bestimmte Speicherstellen in der Zeropage ($00 bis $04) nach anderen Werten als »00« oder »01«. Der Grund für fünf Speicherstellen liegt in der Anzahl der internen Datenpuffer (entspricht jeweils 256 Byte Speicher). Nur von ihnen aus kann die Floppy Daten zum oder vom C64 übertragen. In der Zero-Page ist daher für jeden Puffer eine Speicherstelle (Befehl) reserviert, die dem Betriebssystem sagt, was es auszuführen hat (Jobcode). Zusätzlich stehen diesen Speicherstellen noch jeweils zwei andere gegenüber ($06 bis $0F), mit denen die Spur- und Sektornummer bei Diskettenoperationen mitgeteilt wird:

Puffer	Befehl	Track/Sektor
0	$0000	$0006/$0007
1	$0001	$0008/$0009
2	$0002	$000A/$000B
3	$0003	$000C/$000D
4	$0004	$000E/$000F

Wenn in die Befehlsspeicherstellen ein Jobcode eingetragen wird, führt das Betriebssystem diesen Befehl aus, ohne ihn auf Sinn oder Unsinn zu überprüfen. Das heißt, wenn Sie Track 255/Sektor 90 für Puffer 0 in $0006/$0007 eintragen (+RETURN) und anschließend den Jobcode für Lesen ($80) in $0000 schreiben (+RETURN), versucht die Floppy Track 255 zu lesen, und das geht natürlich nicht, da nur 36 zur Verfügung stehen. Erfolg: Der Schreib-Lese-Kopf kann hängenbleiben und muß von Hand zurückgezogen werden. Also **Vorsicht** bei den Operationen mit Jobcodes:

$80 - Lesen
$90 - Schreiben
$C0 - »Anschlagen« des Kopfes
$D0 - Maschinenprogramme im Puffer ausführen
$E0 - Programm im Puffer ausführen mit Hochfahren des Laufwerks

2. SMON-Extra

... bietet elf zusätzliche Befehle zum SMON. Laden Sie auch hier zuerst die richtige Version für Ihren C64. Danach laden Sie das Generierungs-Programm mit

```
LOAD"SMON+",8
```

Danach geben Sie RUN ein. Nach der Eingabe der Startadresse (49152) wird das neue File generiert. Speichern Sie anschließend die neue Version aus dem SMON auf Ihre Arbeitsdiskette:

```
S"SMON+ $C000"C000 CFFF
```

Die Erweiterung ist jetzt einsatzbereit. Sie bietet folgende Befehle:

Z Start Ende - Zeichensatzdaten

... gibt den Speicherinhalt von Start bis Ende pro Zeile in der 8-Bit-Form aus (09 = ...*.*). Die Zeichen lassen sich zum Ändern überschreiben.

H Start Ende - Sprite-Daten

... entspricht »Z«, nur werden 3 Byte nebeneinander ausgegeben (kleiner Sprite-Editor).

N Start Ende - Bildschirmcode

... interpretiert den Bereich von Start bis Ende als Bildschirmcode und gibt jeweils 32 Zeichen pro Zeile am Bildschirm aus.

U Start Ende - Übersicht des Bildschirmcodes

... wie »N«, aber 40 Zeichen pro Zeile. Die Zeichen lassen sich im Gegensatz zu »N« nur anzeigen, nicht aber überschreiben.

E Start Ende - Füllen mit $00

... füllt den Bereich Start bis Ende mit $00.

Y neu - Anpassen von SMON

... paßt SMON an andere Speicherbereiche an. Dabei muß »neu« zweistellig eingegeben werden. Es wird nur das obere Nibble gewertet. »Y 40« paßt an den Bereich $4000 bis $4FFF in nur drei Sekunden an.

Q Start - Zeichensatz kopieren

... kopiert den Zeichensatz aus dem ROM ins RAM ab Speicherstelle Start.

J - Wiederholen des letzten Befehls

... bringt den letzten Ausgabebefehl zurück auf den Bildschirm.

3. SMON-Illegal

... dissassembliert auch illegale Befehle für den 6510. Dafür entfallen die Diskmonitor-Routinen. Da man sich über den Sinn der illegalen Codes streiten kann (sie funktionieren nicht bei jedem C64) sind sie im Assembler nicht integriert. Laden Sie zuerst die richtige $C000-Version für Ihren C64, dann das Overlay-Programm mit

```
LOAD"NSDIASS",8,1
```

Nach der Eingabe von NEW starten Sie SMON mit »SYS49152«. Jetzt müssen Sie zuerst mit

```
G CFOD
```

die neuen Befehle initialisieren. Danach sollten Sie die neue Version auf Ihre Arbeitsdiskette speichern:

```
S"NS-SMON $C000"C000 CFFF
```

Ihre SMON-Version für illegale Codes ist jetzt einsatzbereit. Wenn Sie jetzt im SMON unser kleines Demo laden:

```
L"ILLDEMO"
```

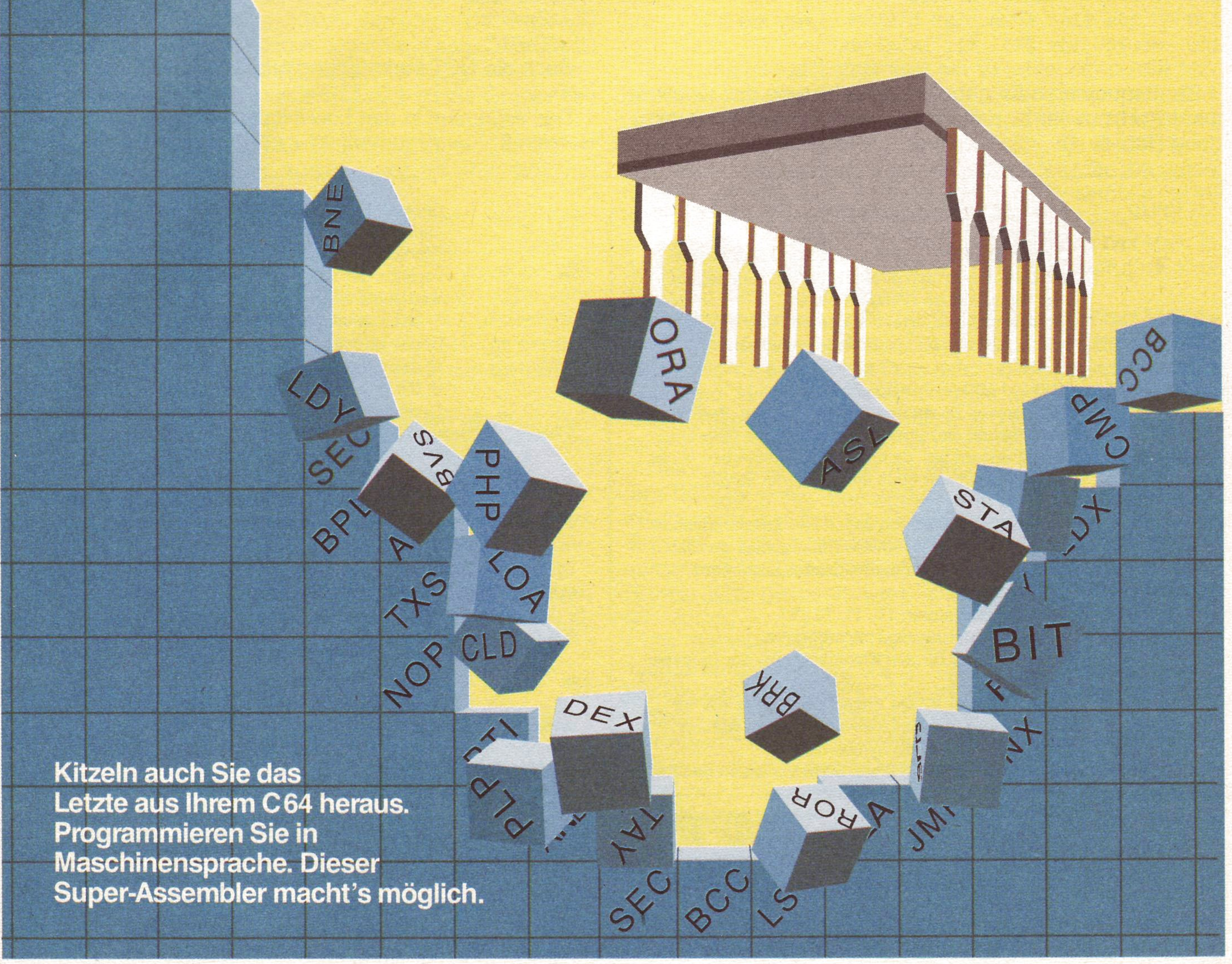
und mit »D4000« dissassemblieren, sehen Sie aufgelistet alle illegalen Codes.

Kurzinfo: SMON

Programmart: Maschinensprache-Monitor
Laden: LOAD"SMON $C000",8,1
Starten: nach dem Laden NEW und SYS 49152
Besonderheiten: Drei unterschiedliche Versionen und drei Erweiterungen
Benötigte Blocks: je 17 Blocks
Programmautoren: Dietrich Weineck/Mark Richter

Hypra-Assembler – Spitzenklasse

Programmieren wie die Profis



Kitzeln auch Sie das Letzte aus Ihrem C64 heraus. Programmieren Sie in Maschinensprache. Dieser Super-Assembler macht's möglich.

von Gerd Möllmann

Ein wichtiges Werkzeug für Maschinensprache-Fans ist der Assembler. Wenn Sie schon immer schnelle Grafiken oder tolle Sounds bewundert haben, finden Sie in Hypra-Ass genau das Richtige. Er ist ein in Maschinensprache geschriebener Drei-Paß-Assembler, d.h. Sie editieren komfortabel in einem eigenen Editor Ihre Quelltexte, danach assemblieren Sie. Hypra-Ass prüft zunächst auf richtige Schreibweise, dann werden Ablageadressen und Tabellen berechnet und in die entsprechenden Speicherzellen abgelegt. Im dritten Paß geschieht (falls gewünscht) die Quelltextaufbereitung.

Sie laden von der beiliegenden Diskette mit:

```
LOAD"HYPRA-ASS",8
```

und starten mit RUN. Hypra-Ass meldet sich mit:

```
BREAK in 0
```

Dieser Text ist kein Fehler, sondern zeigt an, daß alle Funktionen einsatzbereit sind.

Der Quelltext

... wird automatisch in Basic-Programmzeilen abgelegt. Soweit wie möglich werden unnötige Blanks (Leerzeichen) dabei eleminiert. Für die einzelnen Quelltextzeilen gelten dabei folgende Vereinbarungen:

1. Bei der Eingabe einer Zeile wird hinter der Zeilennummer ein Minuszeichen eingegeben.
2. Jede Zeile enthält höchsten einen Assembler-Befehl.
3. Vor einem Assembler-Befehl darf in derselben Zeile höchstens ein Label stehen.

4. Label beginnen direkt hinter dem Minuszeichen.
5. Vor jedem Assembler-Befehl steht mindestens ein Blank.
6. Label und Assembler-Befehl werden mindestens durch ein Blank voneinander getrennt.
7. Ein Label darf nicht allein in einer Zeile stehen.
8. Kommentar wird durch ein Semikolon vom Rest der Zeile getrennt.
9. Reine Kommentarzeilen müssen als erstes Zeichen hinter dem Minuszeichen ein Semikolon haben.
10. Pseudo-Ops (.ba, .eq usw.) können direkt hinter dem Minuszeichen beginnen.

Beispiele:

```
100 -.ba $C000
110 -initialisierung
120 -; reine Kommentarzeile
130 -  lda $14; Kommentar hinter einem Befehl
140 -marke  ldx $15; mit Label davor
```

Zur bequemeren Eingabe und Bearbeitung des Quelltextes stellt Hypra-Ass im Editor insgesamt 25 Befehle zur Verfügung (Tabelle 1).

Rechnungen im Quelltext

Erlaubt sind die vier Grundrechenarten plus Potenzierung, die logischen Operationen NOT, AND und OR, die Vergleiche »gleich«, »kleiner« und »größer«, sowie der Einsatz der Funktionen <(...) und >(...), die das Low- bzw. High-Byte eines Arguments liefern. Die logischen Operatoren und Vergleiche werden wie folgt abgekürzt:

!n! = not
!a! = and
!o! = or
!=! = gleich
!<! = kleiner als
!>! = größer als

Das Ergebnis eines Vergleichs ist -1, falls wahr, 0, falls nicht wahr:

(1!=!2)=0
(1!=!1)=-1

Auch die NOT-Verknüpfung arbeitet wie in Basic:

!n!1=-2

Das Argument in den Low-/High-Byte-Funktionen muß im Bereich 0 < = Argument < = 65535 liegen.

Befehl	Bedeutung
/a 100,10	Automatische Zeilennumerierung. Hier mit der Startnummer 100 und der Schrittweite 10. Die automatische Zeilennumerierung wird ausgeschaltet, indem man direkt hinter dem ausgegebenen Minuszeichen RETURN eingibt.
/o	Re-New eines Quelltextes, der mit NEW gelöscht wurde, falls der Text nicht anderweitig zerstört wurde.
/d ; /d 100 ; /d —100 ; /d 100— ; /d 100-200	Löschen von Zeilen und Zeilenbereichen. Auch für das Löschen einzelner Zeilen sollte man den /d-Befehl verwenden, da man das Minuszeichen hinter der Zeilennummer doch immer wieder vergißt.
/e ; /e 100 ; /e —100 ; /e 100— ; /e 100-200	Formatiertes Listen von Zeilen und Zeilenbereichen. Label, Assembler-Befehle werden gemäß den Tabulatoren übersichtlich untereinander geschrieben.
/t 0,13 ; /t 1,24 ; /t 2,0 ; /t 3,10	setzt die Tabulatoren T0, T1, T2, T3 T0 = Tabulator für Assemblerbefehle T1 = Tabulator für den Kommentar bei der formatierten Ausgabe T2 = Tabulator für die Anzahl der Blanks, die am Anfang einer Ausgabezeile ausgegeben werden T3= Tabulator für die Symboltabelle
/x	Verlassen des Assemblers. Beim Verlassen des Programms wird ein Reset durchgeführt.
/p 1,100,200	Setzen eines Arbeitsbereichs (Page). Hier Bereich 1 von Zeile 100 bis 200, beide einschließlich. Bis zu 30 solcher Arbeitsbereiche sind erlaubt. Die Parameter der Arbeitsbereiche werden im Kassettenpuffer abgelegt.
/ziffer(n)	Formatiertes Listen der Page.
/n 1,100,10	Neu Durchnumerieren einer Page mit Startnummer und Schrittweite.
/f 1,"string"	Suchen einer Zeichenkette in einer Page. Dabei sind im String Fragezeichen als Joker erlaubt. Das Fragezeichen ersetzt ein beliebiges Zeichen. Zu beachten ist jedoch, daß im Quelltext unnötige Blanks entfernt wurden, wie ein Vergleich mit den Befehlen /e und LIST zeigt.
/r 1,"string1", "string2"	Ersetzen von Zeichenketten. String 2 darf nicht leer sein. Überall in der Page wird die Zeichenkette aus String 2 durch die aus String1 ersetzt. Auch beim Ersetzen ist in String 2 das Fragezeichen als Joker erlaubt. Da String 1 leer sein darf, können mit diesem Befehl auch Zeichenketten gelöscht werden.
/u 9000	Setzen des Quelltextstartes (Programmstartes). Normalerweise ist als Startwert die Adresse 7000 eingestellt. Durch Hochlegen des Startes kann man zum Beispiel einen Monitor in dem nun freien Bereich unterbringen.
/b	Anzeige der aktuellen Speicherkonfiguration. Es wird angezeigt: a) der normale Quelltextstart 7000 als Merkhilfe b) der aktuelle Quelltextstart c) das Quelltextende d) die Anzahl der noch verbleibenden Bytes für den Quelltext
/l"name" ; /s"name" ; /v"name" ; /m"name"	Kurzform der Befehle LOAD, SAVE, VERIFY und MERGE
/g 8	Die zugehörige Gerätenummer kann mit diesem Befehl eingestellt werden. Voreingestellt ist das Gerät 8.

Zur Unterstützung des Umgangs mit dem Floppy-Laufwerk 1541 sind drei Befehle implementiert:

Befehl	Bedeutung
/i	— Lesen des Inhaltsverzeichnisses von Floppy ohne Verlust des geschriebenen Quelltextes
/k	— Lesen des Fehlerkanals
/@	— Übermittlung von Befehlen an die Floopy

Diese drei Befehle entsprechen denen des DOS 5.1.

Auch zur Farbgebung des Bildschirms sind zwei Befehle vorhanden, die die Hintergrund- und die Rahmenfarbe setzen.

Befehl	Bedeutung
/ch 0	— Setzen der Hintergrundfarbe
/cr 0	— Setzen der Rahmenfarbe

Nach erfolgter Assemblierung kann nun die erzeugte Symboltabelle mit zwei Befehlen ausgegeben werden:

Befehl	Bedeutung
/!	— Ausgabe in unsortierter Form
/!!	— Ausgabe sortiert

Es werden nur Label ausgegeben, die entweder global oder von der Ordnung Null sind.

Beide Dumps können mit der CTRL-Taste verlangsamt und mit der STOP-Taste angehalten werden.

Mit OPEN... und CMD... können die Dumps an andere Geräte gesendet werden.

Als Ergänzung zum Basic-Befehl PRINT, der aufgrund der Tokenbildung nicht alle Labelnamen verarbeiten kann, kann der Befehl ← verwendet werden.

Basic-Funktionen wie PEEK sind nur über den PRINT-Befehl erreichbar. Die Funktionen <(...) und >(...) sind außerhalb des Quelltextes nur durch ← zu verwenden. Mit dem ←-Befehl kann genau wie im Quelltext gerechnet werden.

Tabelle 1. Die Editor-Befehle von »Hypra-Ass«

Außer Dezimalzahlen sind Hex-Zahlen erlaubt, die durch ein vorangestelltes Dollarzeichen kenntlich gemacht werden:
$C000 = 49152
$10 = 16
$A = 10
Die Hexzahlen können auch in den Basic-Befehlen verwendet werden.

Hypra-Ass-Variable (Label)
Der Wert einer Hypra-Ass-Variablen liegt immer zwischen 0 und $FFFF. Variablennamen dürfen beliebig lang sein, wobei das erste Zeichen des Variablennamens ein Buchstabe sein muß. Weitere Zeichen können Buchstaben, Ziffern oder das Hochkomma sein.

Im Zusammenhang mit der Verwendung von Makros muß zwischen globalen und lokalen Variablen unterschieden werden. Jede Variable erhält beim Anlegen eine sog. Ordnungszahl, die angibt, im wievielten Makroaufruf das Anlegen stattfand. Befindet man sich in keinem Makro, ist die Ordnungszahl dementsprechend Null.

Variable mit unterschiedlicher Ordnungszahl sind trotz gleichen Namens nicht gleich, sondern nur Variable gleicher Ordnungszahl.

Die Konstruktion mittels Ordnungszahlen dient dazu, Fehler durch doppelte Benutzung von Labels auszuschließen.

Andererseits sind aus einem Makro heraus gesehen alle Variablen mit anderer Ordnungszahl als im Makro selbst »unsichtbar«. Um aber bequem Makros in Makros aufrufen und Label verwenden zu können, die in mehreren Makros benutzt werden sollen (etwa Betriebssystemroutinen), gibt es die globalen Variablen.

Globale Variablen sind, wie der Name schon verrät, im Gegensatz zu den lokalen Variablen unabhängig von der Ordnungszahl überall definiert.
Alle Makronamen sind per Definition global.
Alle Variablen sind bei Hypra-Ass redefinierbar gehalten, das heißt alle Variablen können durch eine Wertzuweisung jederzeit verändert werden.

Eine doppelte Benutzung von Variablen wird jedoch durch einen »Label twice«-Error (Tabelle 2) geahndet, da dies zu einem falschen Ergebnis der Assemblierung führen würde.

Die Makros von Hypra-Ass
Makros sind meist kürzere Befehlsfolgen, die im Quelltext häufiger vorkommen, und deshalb unter einem Makro zusammengefaßt werden. Zu jedem Makro gehört ein Name, mit dem es aufgerufen werden kann. An jedes Hypra-Ass-Makro können beliebig viele Parameter übergeben werden, deren aktueller Wert dann bei der Assemblierung im Makro eingesetzt wird. Makros können bei Hypra-Ass an beliebiger Stelle im Quelltext definiert werden. Alle Makronamen sind global, alle Parameter und makrointernen Label sind lokal. Das heißt verschiedene Makros können durchaus Label bzw. Parameter gleichen Namens verwenden.

Ein Beispiel für ein einfaches Makro:
Es wird immer wieder die Befehlsfolge benötigt, Akkumulator und x-Register mit dem Inhalt zweier aufeinanderfolgender Speicherstellen zu laden. Ein Makro dazu könnte folgendermaßen aussehen:

```
100 -.ma ldax (adresse)
110 -     lda adresse
120 -     ldx adresse+1
130 -.rt
```

Dem .ma-Pseudobefehl folgt ein Variablenname, der Makroname, und eine Parameterliste in runden Klammern, falls die Parameter vorhanden sind. In unserem Beispiel ist es ein Parameter, die Adresse der Speicherzelle, die in den Akku soll. Sind mehrere Parameter vorhanden, werden sie durch Komma getrennt. In die Parameter setzt der Assembler bei jedem Aufruf den aktuellen Wert, der im Aufruf steht. Rufe ich also ldax (2) auf, entsteht bei der Assemblierung des Makros die Folge lda 2, ldx 3, entsprechend führt der Aufruf mit ldax (label) zu lda label, ldx label+1.

Die Parameterliste darf in der Definitionszeile eines Makros nur aus einer die Folge von Variablennamen bestehen, während im Aufruf als aktuelle Parameter beliebige Ausdrücke erlaubt sind. Hinter der Definitionszeile mit dem .ma-Pseudo folgt der eigentliche Makroinhalt, also das, was bei einem Aufruf des Makros assembliert werden soll.

Natürlich sind hier nicht nur einfache Befehle wie in unserem Beispiel gestattet. Genausogut können im Makro Verzweigungen und Sprünge ausgeführt werden, es kann bedingt assembliert werden und weitere Makros lassen sich aufrufen. Für die Schachtelung von Makros besteht keine Grenze außer der begrenzten Fassungskapazität des Prozessor-Stacks.

Als Beispiel: Wird ein Makro mit zehn internen Labeln 100mal aufgerufen, ergibt sich schon für die dadurch erzeugten lokalen Label ein Platzbedarf von 7000 Byte.

Sollte irgendwann der Fall eintreten, daß Label und Quelltext zusammen nicht mehr ins RAM passen, erhalten Sie den »too many labels«-Error. Dies ist allerdings mehr ein theoreti-

---

Zusätzlich zu den Fehlermeldungen, die von Interpreterroutinen wie »illegal quantity« oder »syntax« stammen, gibt Hypra-Ass folgende Meldungen aus:
1. **can't number term** — ein Ausdruck kann von Hypra-Ass nicht berechnet werden. Möglicher Grund kann die falsche Abkürzung eines Operators sein.
2. **end of line expected** — bei der Abarbeitung einer Zeile wurde statt des Zeilenendes etwas anderes gefunden.
3. **no mnemonic** — ein Mnemonic kann nicht identifiziert werden.
4. **unknown pseudo** — ein Pseudo-Op wurde falsch abgekürzt.
5. **illegal register** — ein Assemblerbefehl existiert in der gewählten Adressierungsart nicht mit dem gewählten Register.
6. **wrong address** — ein Assemblerbefehl existiert nicht in der gewählten Adressierungsart.
7. **illegal label** — das erste Zeichen eines Labels war kein Buchstabe.
8. **unknown label** — in Pass 2 wurde ein unbekannter Labelname entdeckt
9. **branch too far** — eine Verzweigung führt über eine zu große Distanz.
10. **label declared twice** — ein Labelname wurde zweimal benutzt.
11. **too many labels** — Label und Quelltext passen zusammen nicht mehr in den Speicher.
12. **no macro to close** — die Anzahl der .ma-Anweisungen stimmt nicht mit der Anzahl der .rt-Anweisungen überein.
13. **parameter** — im Makroaufruf stimmt die Parameterliste nicht mit der Parameterliste der Definition überein.
14. **return** — es liegt keine Rückkehradresse auf dem Stack, als eine .rt-Anweisung ausgeführt werden sollte.

Hinzuweisen ist noch auf eine einfache Möglichkeit, den »label twice-error« zu vermeiden:
Legt man eine Makrodefinition um einen beliebigen Block des Quelltextes, so sind alle Label in dem Block automatisch lokal. Auf diese Weise kann schon vorhandener Quelltext in neuen eingefügt werden, ohne daß man sich um doppelt verwendete Labelnamen kümmern muß.

Tabelle 2. Alle Fehlermeldungen auf einen Blick

scher Fall, denn auch bei der Assemblierung von Hypra-Ass selbst wurden trotz extensiver Benutzung von Labels nicht einmal 500 gebraucht. Sie können aber davon ausgehen, daß Ihnen immer mindestens Platz für 1 170 Labels zur Verfügung steht — in den meisten Fällen erheblich mehr.

Selbstaufrufe von Makros sind nicht verboten. Inwieweit eine solche Konstruktion überhaupt sinnvoll sein kann, muß jeder selbst prüfen.

Zurück zur Makrodefinition: Jede Makrodefinition muß unbedingt mit dem Pseudo .rt (return) abgeschlossen sein. Trifft der Assembler bei der Abarbeitung eines Makros auf ».rt«, heißt das für ihn, die Assemblierung hinter dem Aufruf fortzusetzen.

Vor der .ma und .rt-Anweisung dürfen in derselben Zeile keine Label stehen. Die Makrodefinition selbst wird in Pass1 und Pass2 überlesen. Es zählen also nur die Makroaufrufe bei der Assemblierung.

Der Aufruf eines Makros erfolgt durch den Pseudobefehl ..., gefolgt vom Makronamen und der aktuellen Parameterliste in runden Klammern.

Wertzuweisung an Label

Zwei Pseudobefehle stehen zur Verfügung, um Label einen Wert zuzuweisen:

.eq - weist einen Wert zu, ohne die Ordnungszahl des Labels dabei zu verändern.

.gl - erklärt gleichzeitig das Label als global.

Beide Pseudos werden der eigentlichen Wertzuweisung vorangestellt, wie auch LET in Basic:

```
100 -.eq marke = $ffc0
110 -.gl label = $200
```

Bei der Wertzuweisung an Label ist immer der Bereich einzuhalten, in dem ein Labelwert liegen darf (0 bis $FFFF).

Einfügen von Tabellen und Text

Drei Preudo-Ops erleichtern das Einfügen von Tabellen und Text in den Quelltext:

.by - erlaubt das Einfügen von Bytewerten (Werte zwischen 0 und $ff). Einzelne Bytewerte werden durch Komma voneinander getrennt. Auch Strings der Länge 1 sind als Bytewerte erlaubt:

```
100 -.by 0,"a",123,"x",$fa
```

.wo - erlaubt das Einfügen von Adressen (Werte zwischen 0 und $ffff). Mehrere Adressen werden durch Komma voneinander getrennt. Die Adressen werden in der Folge Low-/High-Byte in den Objektcode aufgenommen:

```
100 -.wo marke-1, label*2-1
```

.tx - erlaubt Einfügen von Text in den Quelltext. Die einzelnen Zeichen werden als ASCII-Code im Objektcode aufgenommen:

```
100 -.tx "beispieltext"
```

Immer wenn Byte-Werte im Quelltext erwartet werden (z.B. bei unmittelbarer Adressierung) können Strings der Länge 1 verwendet werden. Ein Befehl lda# " ist damit erlaubt.

Die bedingte Assemblierung

Zur Unterstützung der bedingten Assemblierung bietet Hypra-Ass die Befehlsfolgen IF/THEN/ENDIF und IF/THEN. Außerdem steht ein unbedingter Sprungbefehl zur Verfügung:

.on - entspricht IF/Then aus Basic. Hinter .on folgt ein Ausdruck, ein Komma und ein zweiter Ausdruck. Ist der erste Ausdruck wahr, wird zu der Zeilennummer gesprungen, die der zweite Ausdruck angibt:

```
100 -.on switch !=! 7,400
```

Die Assemblierung wird in Zeile 400 fortgesetzt, wenn switch gleich 7 ist.

.go - ergibt einen unbedingten Sprung zu der hinter .go angegebenen Zeile:

```
100 -.go 1000
```

.if - wird gefolgt von einem Ausdruck. Ist dieser wahr, wird die Assemblierung hinter der .if-Zeile fortgesetzt, bis

.el - gefunden wird. Daraufhin wird

.ei - gesucht und dahinter die Assemblierung fortgesetzt.

Entsprechend erfolgt die Assemblierung von .el bis .ei, falls der Ausdruck hinter .if falsch ist. Wenn .el fehlt, wird direkt hinter .ei fortgefahren.

Auf eine Verschachtelung von IF-Konstruktionen wurde verzichtet. Beispiel:

```
100 -. if switch !=! 6
110 -       lda #0
120 -.el
130 -       lda #2
140 -.ei
```

Wenn switch gleich 6 ist, erhält man lda #0, sonst wird lda #2 erzeugt. Vor den Pseudos .if, .el, und .ei dürfen keine Label in derselben Zeile stehen.

Verkettung von Quelltexten

Mit dem Pseudo .ap (apend) kann ein weiterer Quelltext am Ende des Pass 2 automatisch nachgeladen werden, wobei der Programmzähler aus der vorangegangenen Assemblierung erhalten bleibt.

Hinter .ap muß der Name des nachzuladenden Files in Anführungszeichen stehen.

Eine Besonderheit von Hypra-Ass bildet im Zusammenhang mit verketteten Quelltexten der Pseudo-Opcode .co (common).

Dieser Befehl bewirkt zunächst, daß alle Variablen/Label, die hinter der .co-Anweisung in einer Liste stehen, an den nachgeladenen Teil übergeben werden.

Zweitens bleiben alle Quelltextzeilen bis zur common-Zeile beim Nachladen erhalten. Steht also etwa ein Makro vor der common-Zeile, wird auch das Makro übergeben. Zu beachten ist dabei:

- Es sollten keine Makroaufrufe im common-Bereich stehen, es sei denn innerhalb eines Makros.
- Eine die Startadresse des Objektcodes bestimmende .ba-Anweisung sollte außerhalb des common-Bereiches liegen, damit nach dem Nachladen nicht wieder mit der gleichen Startadresse assembliert wird.
- Wertzuweisungen an Label sollten ebenfalls außerhalb des common-Bereichs sein, um Platz für den nachgeladenen Quelltext zu gewinnen.

Direktes Senden des Objektcodes zur Floppy

Der Pseudobefehl .ob (object), gefolgt vom Filenamen ,p,w (in Anführungszeichen), sendet den erzeugten Objektcode direkt zur Floppy.

Geschlossen wird das so erzeugte Objekt-File durch den Pseudobefehl .en.

Sollte während der Assemblierung ein Fehler entdeckt werden und das Objekt-File nicht schon durch die Hypra-Ass-Fehlerroutine geschlossen worden sein, geben Sie ein:

```
CLOSE14
```

Ausgabe formatierter Listings

.li 1,3,0

sendet ein formatiertes Listing des Quelltextes unter der logischen Filenummer 1 an das Gerät 3 mit der Sekundäradresse 0 (Bildschirm). Die Parameter hinter .li entsprechen denen des OPEN-Befehls. Dadurch wird es auch möglich, das Listing auf eine User-Datei umzuleiten:

```
.li 2,8,2,"test,u,w"
```

Der .li-Befehl muß der erste im Quelltext sein, Zeilen bis einschließlich .li werden nicht ausgegeben. Formatierte Zeilen haben folgendes Format:

```
C000 A0BOCO: 1000 -marke befehl ;kommentar
```

Das Listing des Quelltextes erhält die Kopfzeile »Hypra-Ass Assemblerlisting:«. Die Steuerung der Formatierung erfolgt

mit dem Editor-Befehl »/t«. Bei Zeilen, die Pseudobefehle enthalten, wie .eq..., werden keine Adressen und Opcodes ausgegeben.

.sy 1,3,0

sendet am Ende von Pass 2 die sortierte Symboltabelle. Die Formatierung wird hier durch »/t3,...« gesteuert. Die Labelwerte werden hexadezimal ausgegeben:

```
sprungziel = $FFD2
```

Die Symboltabelle erhält die Kopfzeile »Symbols in alphabetical order«.

.dp t0,t1,t2,t3

setzt aus dem Quelltext heraus die Tabulatoren für:
t0 - Assembler-Befehle
t1 - den Kommentar
t2 - Anzahl der Blanks am Anfang der Ausgabezeile
t3 - Symboltabelle

.st

Nach dem zweiten Pass wird die Meldung »end of assembly«, gefolgt von der Assemblierungsdauer in Minuten, Sekunden und Zehntelsekunden ausgegeben. Dahinter folgt die Zeile »Base = $XXXX last byte at $YYYY«.

Die Zusammenfassung aller Pseudobefehle finden Sie in Tabelle 3.

Um das Editieren der Texte komfortabler zu gestalten, besitzt Hypra-Ass einige neue Eingabefunktionen.

Grundlage dafür blieb dabei der Basic-Editor. Ein Hypra-Ass-Editor wird also genauso eingegeben wie ein Basic-Programm. Allerdings muß hinter der Zeilennummer grundsätzlich ein Minuszeichen eingegeben werden. Es zeigt den Beginn einer Quelltextzeile an. So eingegebene Zeilen werden als ASCII-Zeilen in den Speicher übernommen. Alle überflüssigen Blanks werden dabei entfernt und unmittelbar danach wird die Zeile formatiert am Bildschirm ausgegeben.

Im Gegensatz zu Basic kann ein Zeile nicht durch Eingabe der Zeilennummer gelöscht werden, sie wird mit »/D Zeilennummer« und <RETURN> entfernt. Eine Ausnahme bildet die Zeilennummer »0«. Sie wird weiterhin mit »0« und <RETURN> entfernt. Eine Beschreibung der Editor-Befehle finden Sie in Tabelle 1. Trotzdem einige Anmerkungen:

/A

... bewirkt eine automatische Zeilennummerierung. Bewegen Sie den Cursor nicht aus der Eingabezeile. Wenn Sie über eine andere Zeilennummer geraten und <RETURN> drücken, sieht das Ergebnis sehr seltsam aus. Abhilfe schafft nur Zeile löschen (/D Zeilennummer) und neu eingeben.

/@

... sendet Befehle zur Diskettenstation. So läßt sich beispielsweise mit:

```
/@N:"name",id
```

eine Diskette neu formatieren. »name« muß dabei in Anführungszeichen stehen und ist die Bezeichnung, »id« ist die zweistellige Kennzeichnung.

/P

... legt Arbeitsbereiche fest. Auf den ersten Blick ist der Sinn dieser Einteilung nicht verständlich. Er bietet die Möglichkeit, jedem zusammengehörigen Quelltextteil einen Arbeitsbereich zuzuweisen. Beispiel:

```
/P 1,0,300
```

Dieser Bereich läßt sich dann in unserem Beispiel mit »/1« auflisten. Legt man die Bereiche geschickt an, entfällt das lästige Suchen nach einzelnen Programmteilen.

Auf der beiliegenden Diskette befindet sich Hypra-Ass zusätzlich mit einem komfortableren Editor. Sie laden ihn mit

```
LOAD"HYPRA-ASS .EDI",8
```

und starten mit RUN. Ein Nachteil soll aber nicht verschwiegen werden. Er verlängert das Programm von 25 auf 40 Blocks. Damit steht weniger Platz für Quelltext zur Verfügung. Für kleine bis mittlere Programme reicht der Platz aber allemal. Im Gegensatz zum ursprünglichen Hypra-Ass arbeitet der Editor nun ohne Zeilennummern. Der Quelltext läßt sich beliebig nach oben oder unten scrollen.

Um diese Quelltextbereiche zu verschieben, zu speichern oder zu laden existieren umfangreiche Befehle. Mit speziellen Blocklade- und Blockspeicher-Routinen läßt sich leicht im Laufe der Zeit eine Unterprogrammsammlung anlegen, um sie nach Bedarf in spätere Programme einzubinden.

Nach dem Programmstart meldet sich der neue Editor mit einer Info- und einer Kommandozeile (Abb. 1). Die Infozeile gibt Auskunft über die aktuelle Zeilennummer. Wird dieses File im normalen Hypra-Ass-Editor geladen, entspricht sie der Zeilennummer. In die Kommandozeile gelangen Sie durch <←>. Danach sind die unter Kommando beschriebenen Befehle gültig. Aber auch im Editier-Modus stehen eine Vielzahl von Befehlen zur Verfügung:

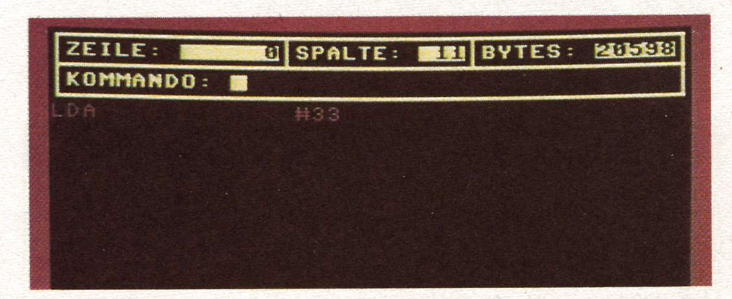
<HOME>

... springt zur weiteren Quelltextbearbeitung in die erste Bildschirmzeile.

1)	.ba $C000	gibt die Startadresse der Assemblierung an. Bei anderen Assemblern heißt dieser Befehl auch org oder * = .
2)	.eq label=wert	weist einem Label einen Wert zu
3)	.gl label=wert	weist einem globalen Label einen Wert zu
4)	.by 1,2,"a"	Einfügen von Byte-Werten in den Quelltext
5)	.wo 1234,label	Einfügen von Adressen in der Folge low/high
6)	.tx"text"	Einfügen von Text als ASCII-Werte
7)	.ap "file"	Verketten von Quelltexten
8)	.ob "file,p,w"	Senden des Objektcodes zur Floppy
9)	.en	Schließen des Objektfiles
10)	.on ausdruck,sprung	bedingter Sprung, wenn Ausdruck wahr
11)	.go sprung	unbedingter Sprung
12)	.if ausdruck	Fortführung der Assemblierung bei ELSE, falls Ausdruck falsch. Ansonsten hinter .if bis zu ELSE oder ENDIF.
13)	.el	Alternative zu den Zeilen, die hinter .if stehen
14)	.ei	Ende der IF-Konstruktion
15)	.co var1,var2	Übergabe von Labeln und Quelltext an nachgeladene Teile
16)	.ma makro (par1,par2)	Makrodefinitionszeile
17)	.rt	Ende der Makrodefinition
18)	...makro (par1,par2)	Makroaufruf
19)	.li, lfn, dn, ba	sendet formatiertes Listing unter der File-Nummer lfn zum Gerät dn mit der Sekundäradresse ba
20)	.sy lfn, dn, ba	sendet formatierte Symboltabelle unter der File-Nummer lfn zum Gerät dn mit der Sekundäradresse ba
21)	.st	beendet die Assemblierung
22)	.dp t0, t1, t2, t3	setzt die Tabulatoren T0, T1, T2, T3 aus dem Quelltext heraus

Vor den Anweisungen 12, 13, 14, 16 und 17 dürfen in derselben Zeile keine Label stehen.

Tabelle 3. Zusammenfassung aller Pseudobefehle



[1] Der komfortable Editor zum »Giga Ass«

<SHIFT HOME>
... springt zur ersten Textzeile.
Mit den Cursor-Tasten läßt
<F1>
... blättert eine Bildschirmseite weiter.
<F2>
... blättert zwei Bildschirmseiten vorwärts.
<F3>
... blättert eine Bildschirmseite rückwärts.
<F4>
... blättert zwei Bildschirmseiten rückwärts.
<F5>
... springt auf den Tabulator 0 und zurück.
<F6>
... ändert die Rahmenfarbe.
<F7>
... ändert die Schriftfarbe.
<F8>
... startet den Assemblier-Vorgang.
<CTRL A>
... markiert die aktuelle Zeile als Blockanfang, mit <CTRL E> wird das Blockende festgelegt. Wird der Cursor zu einer niedrigeren Zeile bewegt, übernimmt die zuerst markierte Zeile das Blockende. Der Anfang muß dann noch mit <CTRL A> markiert werden. Die maximale Blockgröße entspricht 8 KByte.
<CTRL L>
... löscht den markierten Block.
<CTRL C>
... kopiert den markierten Block in den Blockpuffer (ab $E000).
<CTRL E>
... markiert wie schon unter <CTRL A> beschrieben die aktuelle Zeile als Blockende.
<CTRL B>
... fügt einen Block, sofern er im Blockpuffer vorhanden ist, an der aktuellen Cursor-Position ein. Zum Verschieben sollten Sie folgendermaßen vorgehen:
1. Block markieren (<CTRL A>, <CTRL E>)
2. Block kopieren (<CTRL C>)
3. Block löschen (<CTRL L>)
4. Block einfügen (<CTRL B>)
<CTRL D>
... löscht die Cursor-Zeile aus dem Text und vom Bildschirm.
<CTRL I>
... fügt eine Zeile im Programm und auf dem Bildschirm ein. Diese Zeile kann nur nach einer Eingabe z.B. eines REM verlassen werden, ansonsten bleibt der Cursor in dieser Zeile fixiert (auch bei RETURN).
<CTRL K>
... löscht in der aktuellen Zeile den Kommentar nach einem Befehl.
<CTRL R>
... löscht die aktuelle Zeile ab Cursor-Position.
<PFEIL LINKS>
... führt in den Kommandomodus und positioniert den Cursor zur Eingabe in der Kommandozeile.

Der Kommandomodus wird ohne andere Eingaben wieder mit <←> verlassen. Außer diesen Eingaben müssen alle anderen mit <RETURN> bestätigt werden.

Kommandomodus
@ Kommando
... sendet Kommandos an ein Laufwerk. Beispielsweise führt »@V« ein Verify auf der Diskette aus.
BL "Name"
... lädt einen gespeicherten Block in den Blockpuffer.
BS "Name"
... speichert einen markierten Block auf Datenträger, dabei wird der Blockpuffer überschrieben.
GB
... GOTO-Befehl zum Auffinden des Blockanfangs. Die ersten Quelltextzeilen eines Programmes werden am Bildschirm dargestellt.
G100
... listet ab Textzeile 100, sofern diese vorhanden ist.
G"Label"
... sucht die Textzeile, die mit »Label« gekennzeichnet ist und listet ab dieser Position.
I
... listet das Directory einer Diskette auf dem Bildschirm. Es läßt sich mit <F1> vorwärtsblättern.

Wollen Sie einen Disketteneintrag in die Kommandozeile übernehmen, läßt er sich mit <CURSOR aufwärts/abwärts> anwählen und wird mit einer der folgenden Tasten übernommen:
L – übernimmt zum Laden
S – übernimmt zum Speichern
M – übernimmt zum Anhängen an ein bestehendes Programm.
B – übernimmt für Blockload
@ – übernimmt zum Scratchen (Löschen von Diskette).

Danach läßt sich der Text beliebig ändern. Zur Ausführung ist anschließend <RETURN> zu betätigen.
K
... liest den Fehlerkanal eines angeschlossenen Laufwerks und gibt ihn in der Kommandozeile aus.
L"Name"
... lädt das Programm »Name« in den Arbeitsspeicher.
M"Name"
... hängt das Programm »Name« an das im Speicher stehende Programm an. Die Zeilennummern und ihre Schrittweite sind dabei ohne Bedeutung.
S"Name"
... speichert das Programm im Speicher unter »Name« auf Diskette.
N
... entspricht dem Basic-Befehl »NEW«.
O
... holt ein mit »N« gelöschtes Programm zurück.
T0 oder **1,15**
... setzt die von Hypra-Ass bekannten Tabulatoren auf die angegebene Position (sie werden im Hypra-Ass übernommen).
X
... verläßt den Editor und setzt alle Zeiger auf Hypra-Ass.

Mit diesem Editor und Hypra-Ass steht einer komfortablen Assembler-Programmierung nichts mehr im Wege. Für Spezialaufgaben wurde im Sonderheft 53 »Das Beste« der Giga-Ass veröffentlicht. Dieser Assembler bietet zusätzlich einen Generator für EPROM-Module. (gr)

Kurzinfo: Hypra-Ass

Programmart: 3-Pass-Assembler
Laden: LOAD"HYPRA-ASS",8
Starten: nach dem Laden RUN eingeben
Benötigte Blocks: 25
Programmautor: Gerd Möllmann

Kurzinfo: Hypra-Ass mit Editor

Programmart: Assembler mit Scroll-Editor
Laden: LOAD"HYPRA-ASS.EDI",8
Starten: nach dem Laden RUN eingeben
Benötigte Blocks: 40
Programmautor: Gerd Möllmann

Reassembler zu Hypra-Ass

VERWANDLUNGS-KÜNSTLER

von Martin Wehner

Quelltexte sind leicht zu ändern. Hat man keinen zu Verfügung, greift man zu »Hypra-Reass«: Er verwandelt komplexe Maschinen-Programme in durchschaubare Assembler-Listings.

Mit SMON läßt sich ein kleines Programm zwar schnell anschauen, doch wenn Sie Routinen daraus toll finden und in eigenen Werken verwenden wollen, muß Hypra-Reass daraus erst einen Quelltext machen. Er läßt sich beliebig ändern und in eigene Programme einbinden.

Bevor Sie Hypra-Ass starten, müssen Sie nur in wenigen Basic-Zeilen ein paar Anweisungen eintragen. Zuerst sollten Sie allerdings den Reassembler von der beiliegenden Diskette laden:

```
LOAD"HYPRA-REASS",8,1
```

und NEW eingeben. Normalerweise wird jetzt ebenfalls (direkt mit »,8,1« oder im SMON), das zu bearbeitende Maschinenprogramm geladen. Aber wir verwenden zum Üben einen Trick – wir reassemblieren den Reassembler. Dazu geben Sie folgendes kurze Progamm ein:

```
20 ←P $C000
30 ←T $C813,$CAFF
40 ←E 15
```

und tippen im Direktmodus (ohne Zeilennummer):

```
SYS 49152,$C000,$CB00:RUN
```

Gestartet wird anschließend mit <RETURN>. Der Reassembler beginnt nun seine Arbeit und sollte in ca. acht Sekunden mit der Quelltextaufbereitung (17 KByte!) fertig sein. Sehen wir uns aber zunächst die drei neuen Befehle an:

← P Adresse

... markiert Einsprungadressen durch ein Label. Damit sind aus Basic angesprungene SYS-Adressen später im Quelltext leichter auffindbar.

← T Beginn, Ende

... teilt dem Reassembler die Lagen von Tabellen mit. »Beginn« zeigt auf das erste und »Ende« auf das letzte Byte der Tabelle. So markierte Tabellen werden für den Quelltext nicht reassembliert, sondern erscheinen als sog. Hex-Dump (Zeile 10710 bis 12590 im erzeugten Listing).

← E (Byte)

... steht am Ende des Info-Programms. »(Byte)« beeinflußt das Aussehen des Quelltextes (die Klammern dürfen nicht eingetippt werden, sie bedeuten lediglich, daß dieser Wert weggelassen werden kann). Dabei haben einzelne, gesetzte Bits folgende Bedeutung:

0 – markiert alle Label von Zero-Page-Adressen ($00 bis $FF) 3-buchstabig (s. Zeile 100 bis 410). Normal werden Label mit fünf Buchstaben gekennzeichnet.

1 – setzt nach den Befehlen RTS, RTI, BRK und JMP ein Semikolon (Kommentarzeile, z.B. Zeile 1340). So markierte Listing sind etwas übersichtlicher, benötigen aber mehr Speicherplatz.

2 – fügt zu allen Befehlen mit unmittelbarer Adressierung (z.B. LDA #3) einen Kommentar, bei dem der Operand im ASCII-Format ausgegeben wird, wenn er im Wertebereich zwischen 32 und 96 oder 160 und 224 liegt. Ist das nicht der Fall, erscheint ein Punkt (z.B. Zeile 1450).

3 – fügt zwischen je zwei Tabellenzeichen ein Semikolon.

4 – unterdrückt bei Tabellen die ASCII-Anzeige.

5 – kennzeichnet externe Tabellen durch ein vorangestelltes »T« (z.B. TCL000) und externe Label durch »E« (z.B. ECL000). Extern bedeutet hier, daß die Bereiche außerhalb des reassemblierten Bereichs liegen.

6 – läßt den Reassembler selbsttätig nach Tabellen suchen. **Achtung:** Es wird kein Quelltext, sondern ein Informationsprogramm generiert. In diesem sind die Start- und Endadressen aller gefundenen Tabellen. Es empfiehlt sich, diese Werte mit dem SMON zu überprüfen und zu überarbeiten (wie normales Basic). Danach lassen sie sich für die Reassemblierung übernehmen.

7 – schaltet um auf RAM unter ROM. Damit wird die Reassemblierung von Programmen unter dem Basic-Interpreter oder dem Betriebssystem möglich.

SYS 49152,Anfang,Ende(,Position):RUN

... startet den Reassembler und muß im Direktmodus eingegeben werden. »Anfang« definiert die Speicherposition, an dem der Vorgang beginnt, »Ende« entspricht dem Schluß. »Position« kennzeichnet den Beginn an dem das Maschinenprogramm im Speicher liegt. Dieser Wert kann weggelassen werden, wenn es sich an dem Speicherplatz befindet, für den es geschrieben ist. Die Adreßumrechnung wird immer auf den Bereich »Anfang« bis »Ende« bezogen.

Besonderheiten

1. Es sinnvoll, den Reassembler ohne Hypra-Ass zu betreiben, das erzeugte File auf Diskette zu speichern und später unter Hypra-Ass wieder zu laden. Ansonsten ist folgendes zu beachten:

- Verlassen Sie beim Hypra-Ass mit Zusatzeditor den Eingabemodus mit »X«.
- Vergessen Sie beim Info-Programm nicht die Minuszeichen am Anfang jeder Zeile.
- Geben Sie das Info-Programm ohne Leerzeichen ein, da Hypra-Ass nach dem ersten Leerzeichen einen Tabulator einfügt.

2. 3-Byte-Befehle, die bei der Assemblierung als 2-Byte-Befehle interpretiert werden (BIT $A9 $00 = .BY $2C LDA #$00), werden nicht reassembliert. Statt dessen werden die drei Byte mit vorangestelltem .BY-Pseudo-Opcode in den Quelltext eingefügt. Der Befehl wird aber als Kommentar an die entsprechende Zeile angefügt.

3. Hypra-Reass kann bis zu 2700 verschiedene Label verwalten. Die als Einsprungpunkt markierte Adresse darf nicht als Tabellenanfang oder -ende angegeben werden. Im Info-Programm darf keine Adresse doppelt vorkommen.

Natürlich führt auch hier nur Übung zum Erfolg; doch wenn diese Punkte beachtet werden, belohnt Sie Hypra-Reass mit einem fehlerfreien Quelltext. (gr)

Kurzinfo: Hypra-Reass

Programmart: Reassembler
Laden: LOAD"HYPRA-REASS",8,1
Starten: SYS 49152,Anfang,Ende(,Position):RUN
Benötigte Blocks: 12
Programmautor: Martin Wehner

64'er Sonderhefte alle auf einen Blick

Die 64'er Sonderhefte bieten Ihnen umfassende Information in komprimierter Form zu speziellen Themen rund um die Commodore C 64 und C 128. Ausgaben, die eine Diskette enthalten, sind mit einem Diskettensymbol gekennzeichnet.

C 64, C 128, EINSTEIGER



SH 0022: C 128 III
Farbiges Scrolling im 80-Zeichen Modus / 8-Sekunden-Kopierprogramm



SH 0026: Rund um den C64
Der C64 verständlich für Alle mit ausführlichen Kursen



SH 0029: C 128
Starke Software für C 128/ C 128D / Alles über den neuen C 128D im Blechgehäuse



SH 0036: C 128
Power 128: Directory komfortabel organisieren / Haushaltsbuch: Finanzen im Griff / 3D-Landschaften auf dem Computer



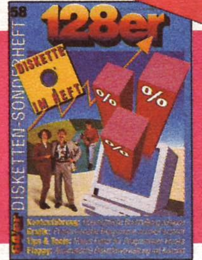
SH 0038: Einsteiger
Alles für den leichten Einstieg / Super Malprogramm / Tolles Spiel zum selbermachen / Mehr Spaß am Lernen




SH 0050: Starthilfe
Alles für den leichten Einstieg / Heiße Rythmen mit dem C 64 / Fantastisches Malprogramm



SH 0051: C 128
Volle Floppy-Power mit "Rubikon" / Aktienverwaltung mit "Börse 128"



SH 0058: 128er
Übersichtliche Buchhaltung zuhause / Professionelle Diagramme



SH 0062: Erste Schritte
RAM-Exos: Disketten superschnell geladen/ Exbasic Level II: über 70 neue Befehle/ Rafinessen mit der Tastatur




SH 0064: 128ER
Anwendungen: USA Journal / Grundlagen: CP/M, das dritte Betriebssystem/ VDC-Grafik: Vorhang auf für hohe Auflösung

GEOS, DATEIVERWALTUNG

SH 0028: Geos / Dateiverwaltung
Viele Kurse zu Geos / Tolle Geos-Programme zum Abtippen



SH 0048: GEOS
Mehr Speicherplatz auf Geos-Disketten / Schneller Texteditor für Geowrite / Komplettes Demo auf Diskette

SH 0059: GEOS
GeoBasic: Großer Programmierkurs mit vielen Tips & Tricks

PROGRAMMIERSPRACHEN



SH 0035: Assembler
Abgeschlossene Kurse für Anfänger und Fortgeschrittene



SH 0040: Basic
Basic Schritt für Schritt / Keine Chance für Fehler / Profi-Tools und viele Tips

ANWENDUNGEN



SH 0031: DFÜ, Musik, Messen-Steuern-Regeln
Alles über DFÜ / BTX von A-Z / Grundlagen / Bauanleitungen



SH 0046: Anwendungen
Das erste Expertensystem für den C 64 / Bessere Noten in Chemie / Komfortable Dateiverwaltung



SH 0056: Anwendungen
Gewinnauswertung beim Systemlotto / Energieverbrauch voll im Griff / Höhere Mathematik und C64

TIPS, TRICKS & TOOLS



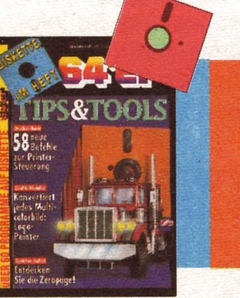
SH 0024: Tips, Tricks& Tools
Die besten Peeks und Pokes sowie Utilities mit Pfiff



SH 0043: Tips, Tricks&Tools
Rasterinterrupts - nicht nur für Profis / Checksummer V3 und MSE / Programmierhilfen



SH 0057: Tips & Tricks
Trickreiche Tools für den C64 / Drucker perfekt installiert



SH 0065: Tips&Tools
Streifzug dirch die Zeropage/ Drucker-Basic: 58 neue Befehle fur Printer-Steuerung/ Multicolorgrafiken konvertieren/ über 60 heiße Tips&Tricks

HARDWARE



SH 0025: Floppylaufwerke
Wertvolle Tips und Informationen für Einsteiger und Fortgeschrittene



SH 0032: Floppylaufwerke und Drucker
Tips&Tools / RAM-Erweiterung des C64 / Druckerroutinen



SH 0047: Drucker, Tools
Hardcopies ohne Geheimnisse / Farbige Grafiken auf s/w-Druckern



SH 0067: Wetterstation:
Temperatur, Luftdruck und feuchte messen/ DCF-Funkuhr und Echtzeituhr/ Daten konvertieren: vom C64 zum Amiga, Atari ST und PC

DTP



SH 0039: DTP, Textverarbeitung
Komplettes DTP-Paket zum Abtippen / Super Textsystem / Hochauflösendes Zeichenprogramm

64'er Magazin
auf einen Blick

Diese 64'er-Ausgaben bekommen Sie noch bei Markt&Technik für jeweils 7,- DM. Die Preise für Sonderhefte und Sammelbox entnehmen Sie bitte dem Bestellcoupon. Tragen Sie Ihre Bestellung im Coupon ein und schicken Sie ihn am besten gleich los, oder rufen Sie einfach unter 089 - 20 25 15 28 an.

9/90: Großer C64-Reparaturkurs / Faszination: Amateurfunk / Neuigkeiten aus der GEOS-Welt / Super-Spiele zum Abtippen

10/90: Bauanleitungen: 5 Wochenend-Projekte / ECOM-das Super-Basic / Test: Die besten Drucker unter 1000 DM / C64-Reparaturkurs

11/90: Bausatztest: Der Taschengeldplotter / Vergleichstest: Drucker der Spitzenklasse / 5 Schnellbauschaltungen

12/90: Abenteuer BTX / Multitasking für C64 / Großer Spieleschwerpunkt / Programmierwettbewerb: 30 000 DM zu gewinnen

1/91: Die Besten Tips&Tricks / Neu: Reparaturecke / Floppy-Flop: Betriebssystem überlistet / Jahresinhaltsverzeichnis

2/91: Sensation: Festplatte für den C 64 / Drucken ohen Ärger / Listing des Monats: Actionspiel "Ignition" / Longplay: Dragon Wars

3/91: Bauanleitung: universelles Track-Display / Alles über Module für den C 64 / Festplatte HD 20 unter GEOS

4/91: Spiele-Schwerpunkt: 100 Tips, News, Tests / Neu: Grafikkurs / Fischer-Baukästen / Bauanleitung: Digitizer

5/91: Ätzanlage unter 50.- DM / GRB-Monitor am C64 / Longplay: Bard's Tale / Reparaturkurs: Die neuen C64 / Piratenknacker

6/91: C64er-Meßlabor: universell Erweiterungsfähig / Test: Pocket-Wrighter 3.0 - Bestes C64 Textprogramm / Listing des Monats: Autokosten im Griff

7/91: Trickfilm mit dem C 64 / Bauanleitung: 1541-Floppy mit Batteriebetrieb / Listing des Monats: Basic-Butler

8/91: Drucker unter 1000 DM / Test: GEO-RAM / Listing des Monats: 80-Farben-Malprogramm / Longplay: Secret of the Silver Plate

09/91: Joystick im Test / Die üblen Tricks mit Raubkopien / Die besten Drucker unter 1500 DM / Mit großem Spieleteil

10/91: 100 besten Tips&Tricks / Listing: Fraktal-Programm / C-64-Meßlabor: komfortables Kontrollmodul

GRAFIK



SH 0020: Grafik
Grafik-Programmierung / Bewegungen



SH 0045: Grafik
Listings mit Pfiff / Alles über Grafik-Programmierung / Erweiterungen für Amica-Paint



SH 0055: Grafik
Amica-Paint: Malen wie ein Profi / DTP-Seiten vom C64 / Tricks&Utilities zur Hires-Grafik

SH 0063: Grafik
Text und Grafik mischen ohne Flimmern / EGA: Zeichenprogramm der Superlative / 3 professionelle Editoren



SH 0068: Anwendungen
Kreuzworträtsel selbstgemacht/ Happy Synth: Super-Synthesizer für Sound-Freaks/ Der C64 wird zum Planetarium/ Sir-Compact: Bit-Packer verdichtet Basic- und Assemblerprogramme

SPIELE



SH 0030: Spiele für C 64 und C 128
Spiele zum Abtippen für C 64/ C 128 / Spieleprogrammierung



SH 0037: Spiele
Adventure, Action, Geschicklichkeit / Profihilfen für Spiele / Überblick, Tips zum Spielekauf



SH 0042: Spiele
Profispiele selbst gemacht / Adventure, Action, Strategie



SH 0049: Spiele
Action, Adventure, Strategie / Sprites selbst erstellen / Virenkiller gegen verseuchte Disketten



SH 0052: Abenteuerspiele
Selbstprogrammieren: Von der Idee zum fertigen Spiel / So knacken Sie Adventures



SH 0054: Spiele
15 tolle Spiele auf Diskette/ der Sieger unseres Programmierwettbewerbs: Crillion II/ ein Cracker packt aus: ewige Leben bei kommerziellen Spielen



SH 0060: Adventures
8 Reisen ins Land der Fantasie - so macht Spannung Spaß



Top Spiele 1
Die 111 besten Spiele im Test/ Tips, Tricks und Kniffe zu heißen Games/ Komplettlösung zu "Last Ninja II"/ große Marktübersicht: die aktuellen Superspiele für den C64



SH 0061: Spiele
20 heiße Super Games für Joystick-Akrobaten/ Cheat-Modi und Trainer POKEs zu über 20 Profi-Spielen/ Krieg der Kerne: Grundlagen zur Spielerprogrammierung



SH 0066: Spiele
15 Top-Spiele mit Action und Strategie/ Mondlandung: verblüffend echte Simulation und Super-Grafik/ High-Score-Knacker: Tips&Tricks zu Action-Games

BESTELLCOUPON

Ich bestelle ____ 64er Sonderhefte Nr. ____________ ____ DM

zum Preis von je: 14,- DM (Heft ohne Diskette) ____ DM
16,- DM (Heft mit Diskette) ____ DM
9,80 DM (SH "Top Spiele 1") ____ DM
24,- DM (für die Sonderhefte 0051/ 0058/ 0064) ____ DM

Ich bestelle ____ 64er Magazin Nr. ____ / ____ / ____ ____ DM
zum Preis von je 7,- DM

Ich bestelle ____ Sammelbox(en) ____ DM
zum Preis von je 14,- DM

Gesamtbetrag ____ DM

Ich bezahle den Gesamtbetrag zzgl. Versandkosten nach Erhalt der Rechnung.

Name, Vorname

Straße, Hausnummer

PLZ, Wohnort

AC 26 10

Telefon (Vorwahl) ____________ Ich erlaube Ihnen hiermit mir interessante Zeitschriftenangebote auch telefonisch zu unterbreiten (ggf. streichen).

Schicken Sie bitte den ausgefüllten Bestellcoupon an: **64er Leserservice, CSJ, Postfach 140 220, 8000 München 5, Telefon 089/ 20 25 15 28**

Die 64 KByte Speicher des C64 sind für den Assembler-Programmierer ein Abenteuer-Spielplatz zum Austoben. Selbstverständlich kommen dabei auch viele nützliche Utilities heraus, wie Ihnen unsere kleine Sammlung kurzer, aber effektiver Programme beweist.



Textausgabe in Maschinensprache

Jeder Assembler-Programmierer kennt die Betriebssystem-Routine zur Ausgabe beliebiger Texte oder Zeichen auf dem Bildschirm: $FFD2 (BSOUT). Man muß den Akku mit dem gewünschten Character laden und das Unterprogramm im Kernel des C64 per JSR (Jump Subroutine) anspringen. Da es recht umständlich ist, pro übertragenem Zeichen einen eigenen LDA-Befehl loszulassen, bringt man den Text oder die gewünschten Zeichen (auch Steuercodes und Blockgrafikzeichen sind möglich!) in einen freien Speicherbereich und richtet z.B. die Routine darauf. Das letzte Textbyte sollte eine Null sein, um eine komfortable Schleife aufbauen zu können. Wenn's nicht mehr als maximal 255 Byte sind, die man ausgeben lassen will (gespeichert ab $C010), könnte das entsprechende Assembler-Programm so aussehen (es läßt sich mit SMON eingeben, Startadresse $C000):

```
A C000 A2 00    LDX #$00
;Zählvariable auf >>0<<
A C002 BD 00 C1 LDA $C010,X
; aktuelles Byte aus Text
; bei $C010 laden
A C005 20 D2 FF JSR $FFD2
; mit BSOUT ausgeben
A C008 E8       INX
; Zähler erhöhen
A C009 C9 00    CMP #$00
; Nullbyte im Akku? dann
; Textende
A C00B D0 F5    BNE $C002
; nein, weiter im Text
A C00D 60       RTS
; Ende, zurück ins Basic
:C010 44 41 53 20 36 34 27 45
:C018 52 2D 53 4F 4E 44 45 52
:C020 48 45 46 54 20 49 53 54
:C028 20 53 55 50 45 52 21 00
; Textdaten im Bereich $C010
```

Laden Sie das entsprechende Demoprogramm mit:

```
LOAD "BSOUT",8,1
```

Geben Sie NEW ein und starten Sie das Programm mit »SYS 49152«.

Noch komfortabler geht's mit der Systemroutine STROUT $AB1E im Basic-Interpreter. Die Textdaten belassen wir im Bereich ab $C010. Jetzt sieht das Assembler-Programm so aus:

```
A C000 A9 10    LDA #$10
; Lowbyte Text in Akku
A C002 A0 C0    LDY #$C0
; Highbyte Text ins y-Reg.
A C004 20 1E AB JSR $AB1E
; Sprung zur Stringausgabe
A C007 60       RTS
; zurück ins Basic
:C010 s. oben!
```

Diese Routine ist 6 Byte kürzer und erheblich schneller. Die Adresse des Textbereichs muß in Low- und High-Byte zerlegt werden. Die beiden Werte kommen in den Akku und das y-Register. Auf der Diskette zu unserem Sonderheft heißt das Beispielprogramm »Strout«. Es gelten die Lade- und Startanweisungen von »Bsout«.

Beide Versionen zur Textausgabe sind zwar bedeutend schneller als der PRINT-Befehl in Basic, aber ein Manko bleibt: Assembler-Routine und Textspeicherbereich liegen oft viele Bytes voneinander getrennt.

Der C128 z.B. kennt die Betriebssystem-Routine PRIMM (Print Immediate). Damit läßt sich Text ausgeben, der unmittelbar hinter dem Sprungbefehl zu dieser Routine steht. Das Assembler-Programm »Outtext« für den C64 auf beiliegender Diskette verfolgt dasselbe Prinzip. Laden Sie das Programm mit:

```
LOAD "OUTTEXT",8,1
```

und geben Sie NEW ein, jedoch keinen SYS-Befehl! Es liegt bei $8000, kann aber im Speicher verschoben werden (relokatibel). Hier das kommentierte Assembler-Listing:

```
A 8000 68       PLA
; letzten Wert vom Stapel
; in Akku holen
A 8001 18       CLC
; Carry-Flag löschen
A 8002 69 01    ADC #$01
; Akku um >>1<< erhöhen
A 8004 85 A7    STA $A7
; in Adresse 167 schreiben
A 8006 68       PLA
; nächsten Wert vom Stapel
; in Akku bringen
A 8007 69 00    ADC #$00
A 8009 85 A8    STA $A8
A 800B A0 00    LDY #$00
A 800D B1 A7    LDA ($A7),Y
; Bytes lesen
A 800F F0 0C    BEQ $801D
A 8011 20 D2 FF JSR $FFD2
; und ausgeben
A 8014 E6 A7    INC $A7
A 8016 D0 02    BNE $801A
A 8018 E6 A8    INC $A8
A 801A 18       CLC
A 801B 90 EE    BCC $800B
A 801D A5 A8    LDA $A8
A 801F 48       PHA
A 8020 A5 A7    LDA $A7
A 8022 48       PHA
; aktuellen Programmzähler
; auf Stapel legen
A 8023 60       RTS
```

Nachdem diese Routine im Speicher installiert ist, kann sie von anderen Maschinensprache-Programmen mit »JSR $8000« aufgerufen werden. Ein Beispiel:

```
A C000 20 00 80 JSR $8000
M C003 41 42 43 44 45 46 00
A C00A RTS (oder weiter im
Programm)
```

Der Textbereich muß ebenfalls mit einem Nullbyte enden. Das Programm macht sich die Eigenheit der Stack-Verarbeitung des C64 zunutze: Bei jedem JSR-Befehl wird die aktuelle Adresse des Programmzählers in Form von High- und Low-Byte auf den Stapel gelegt. Der Computer muß wissen, wohin er nach dem Unterprogramm springen soll. Das Programm liest die unmittelbar dahinterliegenden Bytes ab $C003 und gibt sie auf dem Bildschirm aus, bis das Nullbyte erreicht ist. Erst danach führt er seinen RTS aus.

(Harald Gasch/bl)

Geheimnisse beim Rasterzeilen-Interrupt

Außer Rand und Band

Kaum bekannt ist die Funktion der Speicherstelle $3FFF (16383): Schaltet man per Raster-Interrupt den unteren Teil des Bildschirmrahmens ab, muß in $3FFF der Wert »0« stehen, damit keine störenden, schwarzen Streifen auftreten. Pro Rasterzeile wird das Bitmuster dieser Adresse 40mal auf den Bildschirm gebracht. Befindet sich darin ein Nullbyte, ist nichts zu sehen. Ändert man aber das Bitmuster pro Rasterzeile, lassen sich interessante Effekte im Bildschirmrahmen erzeugen. Und das alles ohne Sprites!

Laden Sie unser Demo-Programm mit:

```
LOAD "RAND",8
```

und starten Sie es mit RUN. Im Programm ist die Maschinensprache-Routine als DATAs in den Zeilen ab 32000 integriert. Sie erledigt folgende Aufgaben:

Zunächst trifft sie alle Vorbereitungen für den Raster-Interrupt. Der obere und untere Bildschirmrand wird ausgeschaltet. Erreicht der Rasterstrahl den unteren Rand, verharrt das Programm in einer Schleife, bis der Strahl wieder den beschreibbaren Teil des Bildschirms einholt. Während dieses Vorgangs schreibt das Programm Daten in die Adressen $3FFF und $D021. Gesteuert wird es durch eine Verzögerungsschleife: Die Übertragung der Daten findet jedesmal bei Erreichen einer neuen Rasterzeile statt. Die jeweiligen Inhalte der Speicherstellen $3FFF und $D021 ändern das Bildmuster und die Hintergrundfarben. Die Bits in $3FFF sind dafür verantwortlich, wie der Bildrand aussieht: In jeder Rasterzeile werden sie 40mal nebeneinander gezeigt. Aus Zeitgründen kann man den Wert in der Rasterzeile nicht ändern, die nächste kann aber mit einem völlig neuen Bitmuster beschrieben werden. Das ergibt den reizvollen Effekt. Ein anderer Programmteil bewegt die Datenfelder, so daß die Muster im Bildschirmrahmen gescrollt werden.

Das Assembler-Programm wurde in den Interrupt eingeklinkt. Beachten Sie die Erläuterungen des Programmautors auf dem Bildschirm. Erst die Tastenkombination <RUN/STOP RESTORE> macht dem munteren Treiben ein Ende. Mit »SYS 36864« läßt sich das Rahmenmuster wieder einschalten. SYS 36953 deaktiviert das Utility.

Dieser Trick klappt auch in den anderen VIC-Bänken des C64: Er bezieht sich generell immer auf die letzte Adresse des aktuellen 16-KByte-Blocks, den der VIC-Chip überblickt. Es funktioniert auch bei $7FFF,$BFFF und $FFFF.

(Steffen Goebbels/bl)



[1] 136 Farben mit dem C64 im Interlace-Modus

136 Farben?

Das ist kein Druckfehler: »Farbdemo« zeigt gleichzeitig 136 Farben auf dem Bildschirm. Auch hier arbeitet der Programmautor kräftig mit dem Rasterzeilen-Interrupt. Das Monitorbild wird 50mal pro Sekunde aus zwei gleichen Halbbildern aufgebaut. »Farbdemo« erzeugt dagegen zwei verschiedene Halbbilder: z.B. auf dem ersten einen roten, auf dem zweiten einen blauen Balken. Aufgrund der hohen Bildwiederhol-Frequenz mischen sich die Farben: Dem Betrachter erscheint der Balken nun violett.

Die beiden Halbbilder sind zwei Bitmaps. Die eine zeigt 16 senkrechte, die andere 16 waagrechte Farbbalken. In Tabelle 1 ist ein Beispiel erläutert.

Farbdemo

Bitmap 1 enthält die waagrechten, Bitmap 2 die senkrechten Farbbalken.

Bitmap 1	Bitmap 2	Bildschirm
rot-rot	rot-blau	rot-violett
blau-blau	rot-blau	violett-blau

Tabelle 1. Beispielfarben für Bitmaps und Bildschirm: »Farbdemo«

Laden Sie unser Demo-Programm von der Diskette zum Sonderheft:

```
LOAD "FARBDEMO",8,1
```

Geben Sie NEW ein und starten Sie das Programm mit »SYS 49152«.

Durch einen Raster-Interrupt werden die Bitmaps 50mal pro Sekunde vertauscht. Dadurch flimmert der Bildschirm etwas (Interlace-Modus). Am unteren Bildschirm erscheinen drei verschiedene Farbbalken (Abb.1). Mit den Tasten <,> und <.> (Komma und Punkt) lassen sich die Farben der oberen und unteren Fläche ändern. Der mittlere Balken zeigt jetzt die Mischfarbe. Damit kann man feststellen, wie stark die Farbmischung flimmert und eine andere wählen.

Das Programm belegt den Speicherbereich von $C000 bis $C23A. Wem der Text der Bildschirm-Ausgabe nicht gefällt, kann ihn ab Adresse $C1E8 seinen Wünschen anpassen. (bl)

Poppiger Screen mit Pep

Viele Programme gibt's, die den Bildschirm in buntschillernde Bereiche einteilen. Die meisten haben einen Nachteil: Lage, Größe und Anzahl der Farbunterteilungen sind nicht variabel genug. Außerdem wird der Bildschirm bis in den letzten Winkel aufgeteilt: Es ist also nicht möglich, z.B. nur eine einzelne Schriftzeile farbig zu unterlegen. »Screenmanager« packt die Sache anders an: Das Utility definiert Zonen anhand einer Tabelle ab Adresse $C094. Sie enthält als ersten Parameter die Rasterzeile -1, in der die erste Zone beginnen soll (sinnvoll sind nur Zahlen von 49 bis 250). Ab dieser Rasterzeile legt das Programm die übrigen Bereiche an. Es folgt die Farbe des ersten Bereichs (Werte von 0 bis 15), anschließend die Breite (wieder in Rasterzeilen, von 1 bis 250). Bei »250« wird allerdings der gesamte Bildschirm umgefärbt. Nach diesem Schema kann man beliebig viele Bereiche anhängen: Farbe, Breite, nächste Farbe usw. Um die Bytefolge zu beenden, hängt man eine Zahl an, die größer als »127« ist. Dahinter läßt sich jetzt eine weitere Farbzone nach demselben Muster einrichten oder die Liste mit »0« beenden. Unsere Tabelle 2 zeigt ein Beispiel, wie die entsprechende Speicherbelegung aussehen könnte.

Auf der Diskette finden Sie drei Dateien zu »Screenmanager«:

- SCRNMANAGER.OBJ, die Maschinensprache-Routine,
- SCRNMANAGER.SRC, der Assembler-Quellcode, den Sie mit »Hypra-Ass« anzeigen oder editieren können,
- SCRNMANAGER-DEMO, ein Basic-Programm, das die Vorzüge von »Screenmanager« verdeutlicht. Laden Sie es mit:

```
LOAD "SCRNMANAGER-DEMO",8
```

und starten Sie es mit RUN. Die Assembler-Routine wird nachgeladen und initialisiert, unmittelbar danach sehen Sie interessante Effekte auf dem Monitor (Abb.2). Den Programmtext in den Zeilen 110 bis 170 können Sie beliebig verändern, ebenso die Farb- und Zonendaten ab Zeile 801.

Nach dem Start mit »SYS 49152« (im Demo-Programm Zeile 270) läuft die Routine im Raster-IRQ zusammen mit jedem beliebigen Basic-Programm. Die normale Hintergrundfarbe muß allerdings in $C072 gespeichert werden (statt in $D021).

»Screenmanager« (Zonentabelle)

Hinweise: Die Tabelle kann beliebig angelegt werden. Farbzonen dürfen sich nicht überschneiden und müssen in der Reihenfolge im Speicher stehen, in der sie auf dem Bildschirm erscheinen. Die Liste darf nicht mehr als 256 Werte enthalten!
Beispiel: Ab den Rasterzeilen 100 und 150 sollen zwei mehrteilige Farbzonen definiert werden.

Adresse (dez.)	Wert
49300	99 (Beginn der 1. Zone minus »1«)
49301	2 (1. Farbe = rot)
49302	10 (zehn Zeilen breit)
49303	5 (2. Farbe = grün)
49304	10 (zehn Zeilen breit)
49305	6 (3. Farbe = blau)
49306	8 (acht Zeilen breit)
49307	255 (Endekennung der 1. Zone)
49308	149 (Beginn Zone 2 minus »1«)
49309	(1. Farbe = gelb)
...	(usw., wie oben)
49314	5 (letzter Bereich, Breite: fünf Zeilen)
49315	255 (Endekennung letzte Zone)
49316	0 (Ende der Tabelle)

Tabelle 2. In dieser Reihenfolge werden die Parameter bei »Screenmanager« in einer Liste erfaßt

In der Zeropage braucht das Programm die Speicherstelle $FB. Beachten Sie: Je größer die definierten Zonen, desto langsamer läuft das Hauptprogramm! Wenn Sie während des Programm-Ablaufs die Zonenliste ab $C094 ändern (z.B. mit POKEs), überschlagen sich die Effekte: Blinken, Farbscrolling, Bewegung usw. (K. Kähler/bl)

Unverrückbar

Gerade Basic-Programmierer haben's schon oft erlebt: Auf dem Bildschirm steht das Bedienungsmenü eines Programms. Durch die Ausgabe einer überlangen Liste scrollt sie plötzlich nach oben und ist spurlos verschwunden. Als geübter Assembler-Programmierer baut man hier vor: Unser Utility »Kopfzeilen« plaziert am oberen Bildschirmrand drei Statuszeilen, die sich durch nichts von dieser Stelle bewegen lassen. Dieser Trick funktioniert wieder nur mit dem Rasterzeilen-Interrupt. Dadurch ist es nicht nötig, das Betriebssystem ins RAM zu kopieren (wie's viele andere Statuszeilen-Programme machen). Das RAM unterm ROM bleibt frei und kann für nützlichere Zwecke verwendet werden.

Das Utility belegt den Speicher von $C000 bis $C095. Dazu braucht es noch 240 Byte Zwischenspeicher zusätzlich.

Laden Sie das Programm mit:

```
LOAD "KOPFZEILEN",8,1
```

Mehrere Einstell- und Initialisierungsmodi stehen zur Verfügung. Sie sollten die Befehle in dieser Reihenfolge eingeben:

POKE 49161,zl: legt die gewünschte Zahl der Statuszeilen fest. ZL darf Werte zwischen »1« und »3« besitzen. Achtung: Das Programm erkennt keine falschen Eingaben!

SYS 49158: rettet die oberen Zeilen inkl. Farbinformation, die mit dem vorangegangenen POKE-Befehl eingestellt wurden, in den Zwischenspeicher.

SYS 49152: startet das Utility. Bei jedem Durchlauf des Elektronenstrahls überträgt das Programm den Inhalt des Zwischenspeicher in Bildschirm und Farb-RAM. Das erweckt die Illusion, die Zeilen wären unzerstörbar und würden sich nicht von der Stelle bewegen.

SYS 49155: schaltet das Hilfsprogramm ab.

Auf der Diskette zu diesem Sonderheft befindet sich ein Beispielprogramm:

```
LOAD "DEMO.KOPFZEILEN",8
```

Starten Sie es mit RUN und betrachten Sie die Bildschirmausgabe: Die obersten drei Zeilen bleiben, wo sie sind! Das Basic-Listing gibt Programmierern wertvolle Hinweise, wie man das Utility in eigene Programme einbaut (z.B. Datenverwaltungen).

Mit <RUN/STOP RESTORE> stellen Sie den Demo-Durchlauf ab.



[2] »Screenmanager« verwaltet Rasterzeilen

Basic-Erweiterung – selbstgemacht

Basic 2.0 ist bestimmt nicht der Weisheit letzter Schluß. Viele komfortable Anweisungen fehlen, die für andere Basic-Dialekte eine Selbstverständlichkeit sind. Doch auch für Assembler-Einsteiger sollte das kein Problem sein: Man integriert zusätzliche Basic-Befehle, die neben den anderen Anweisungen des Basic-Interpreters laufen. Unser simples Beispiel »Disk-Basic« auf der Diskette zu diesem Heft zeigt, wie relativ einfach neue Basic-Befehle geschaffen werden. Dazu muß man sich nur eines kleinen Tricks bedienen: Man klinkt sich in die Routine $A7E4 ein, die Basic-Befehle interpretiert und auf die entsprechende Anfangsadresse weist, die mit dieser Anweisung ausgeführt werden sollen. Derartige Betriebssystem-Routinen werden dem Computer durch Vektoren übermittelt (in unserem Fall: $0308/0309), die solche Einsprungadressen als Low- und High-Byte speichern. Wenn Sie nun einen »Wedge« (Keil) vor die Routine $A7E4 schieben, müssen Sie die Vektoren umpolen: Low- und High-Byte müssen auf das Assembler-Programm zeigen, das neue Basic-Befehle erzeugt, die richtige Schreibweise (Syntax) überprüfen und entsprechend reagieren – war's ein neuer Befehl, verzweigt das Programm zur neuprogrammierten Routine des zusätzlichen Basic-Befehls, andernfalls macht es mit der internen Routine weiter.

Unser Beispielprogramm »Disk-Basic« ist nach diesen Richtlinien aufgebaut. Laden Sie es mit:

```
LOAD "DISK-BASIC",8,1
```

Anschließend tippen Sie NEW ein, betrachten das Assembler-Listing im SMON oder starten das Utility mit »SYS 49152«. Es erweitert das Basic 2.0 um sechs komfortable Diskettenbefehle: DLOAD, DSAVE, DVERIFY, DPRINT, DERROR und DLIST. Dieses Programmprojekt ist recht einfach zu realisieren: Alle neuen Befehle beginnen mit »D«, die restlichen Befehlsbuchstaben stimmen mit Anweisungen überein, die der Basic-Interpreter sowieso kennt. Bei der Befehlseingabe werden sie automatisch in Tokens umgewandelt (ein Byte als Kurzform der Basic-Befehle).

Der Startbefehl »SYS 49152« macht nichts anderes, als die Vektoren-Inhalte in $0308/0309 aufs eigene Erweiterungsprogramm zu richten, das bei $C00B beginnt. Bei jeder Direkteingabe einer Basic-Anweisung wird zunächst in diesem Maschinensprache-Programm überprüft, ob der Befehl mit »D« beginnt. Trifft das nicht zu, macht der Computer ganz normal im Original-Betriebssystem weiter. Dazu verwendet man die CHRGET-Routine ab $0073. Ist der erste Buchstabe des Basic-Befehls ein »D«, springt das Programm zur Adresse $C015. Hier überprüft es, ob bestimmte Tokens folgen:

$93 für LOAD, $94 = SAVE, $95 für VERIFY, $99 für PRINT und $9B = LIST. Der neue Befehl DERROR ist nicht mit Tokens zu erreichen (außer OR), die restlichen Buchstaben müssen per Tabelle eingelesen werden. Die entsprechenden Maschinensprache-Routinen zum Laden, Speichern usw. werden durchs Programm initialisiert:

- DLOAD: $C04E,
- DSAVE: $C061,
- DVERIFY: $C04B,
- DPRINT: $C045,
- DLIST: $C048,
- DERROR: $C02D.

Das Programm ersetzt umständlichere Anweisungen des Basic 2.0:

```
DLOAD "(Programmname)" =
LOAD "(Programmname)",8
DSAVE "(Programmname)" =
SAVE "(Programmname)",8
DVERIFY "(Programmname)" =
VERIFY "(Programmname)",8
```

DPRINT macht's möglich, beliebige Diskettenbefehle ohne OPEN- und CLOSE-Anweisungen an die Floppy zu senden (z.B. NEW, SCRATCH, RENAME usw):

```
DPRINT "N:TESTDISK,ID" =
OPEN 1,8,15,"N:TESTDISK,ID":
CLOSE 1
```

formatiert z.B. eine Diskette und gibt ihr den Namen »Testdisk«.

DLIST zeigt das aktuelle Directory auf dem Bildschirm, ohne ein Basic-Programm im Speicher zu löschen.

DERROR liest den Fehlerkanal der Floppy, gibt die Meldung aus und stellt das Blinken der Floppy-LED ab.

Alle neuen Befehle (außer DERROR) lassen sich wie gewohnt abkürzen. Voraussetzung: Es muß immer das »D« davorstehen. Das Hilfsprogramm benötigt allerdings eine Diskettenstation mit der Geräteadresse »8«. (bl)

Holzauge, sei wachsam!

Welcher Programmier (egal, ob in Basic oder Assembler) kennt diese Situation noch nicht? Das Programm gibt keinen Mucks mehr von sich. Ist es abgestürzt oder hängt es in irgendeiner Berechnungsschleife?

Das Programm »Freemem 53100« zeigt interruptgesteuert den Stack-Pointer und den freien Basic-Speicher (als 16-Bit-Hexadezimalzahl) in der rechten, oberen Bildschirmecke. Vor allem die Anzeige des Stackpointers, die beim Programmablauf ständig wechselt, ist eine tolle Sache zum Testen kritischer Programme (z.B. viele GOSUBs, Schleifen usw.). Außerdem hat man ständig den noch verbleibenden Restspeicher des RAM im Blick oder kann daraus ersehen, wann der Computer eine »Garbage Collection« (Beseitigung von Stringmüll) ausführt.

Laden Sie das Utility mit:

```
LOAD "FREEMEM 53100",8,1
```

und starten Sie es mit »SYS 53100«, nachdem Sie NEW eingegeben haben. <RUN/STOP RESTORE> setzt die Interrupt-Zeiger ($0314/0315) wieder auf den angestammten Wert $EA31 und schaltet das Programm ab.

(H. Müller-Zauleck/bl)

Vertauschte Bildschirme

Wer in seinen Programmen Windows (Ausgabefenster) benutzt, braucht diese Routine: »Screen-Copy«. Damit läßt sich der aktuelle Bildschirminhalt in den Bereich ab $C000 retten. Jetzt kann das Window erscheinen (z.B. Menüs oder Benutzerhinweise) und darf den vorher zwischengespeicherten Bildschirminhalt getrost überschreiben. Der nächste Aufruf unserer Routine bringt den geretteten Bildbereich wieder an die angestammte Stelle. Mit Maschinensprache geht das in Sekundenbruchteilen. Laden Sie das Programm mit:

```
LOAD "SCREEN COPY",8,1
```

Es belegt den Speicherbereich des C64 von $033C bis $0378 im Kassettenpuffer. Die eigentliche Datenübertragungsroutine beginnt bei Adresse $0364. Low- und High-Byte der Start- bzw. Zieladresse werden in temporären Zwischenspeicherstellen abgelegt.

Nach der Eingabe von NEW stehen zwei Bedienungsmöglichkeiten zur Verfügung, die vom Inhalt der Speicherstelle $02A8 (680) abhängig sind:

Bildschirm sichern: POKE 680,0: SYS 828
Geretteten Bildschirm holen: POKE 680,1: SYS 828

Das Programm kann zwar inverse Bildschirmzeichen übertragen und wieder zurückholen, aber keine Farbinformationen.

(Sascha Michalek/bl)

Raffinierte Vergleichsweise

Eifrige Programmierer verwenden die VERIFY-Funktion des Floppy-DOS immer dann, wenn z.B. ein Basic-Programm geändert und auf Diskette zurückgespeichert wurde. Übertragungsfehler bzw. Abweichungen vom Original im Speicher werden sofort offensichtlich: Es erscheint die Meldung »Verify Error«. Das war's schon. Kein Hinweis darauf, wo sich die differierenden Bytes im Programm befinden! Das Utility »Verify-Master V1« ist da viel genauer: Es vergleicht die Daten im Speicher mit denen auf Diskette – und dazu bedeutend schneller als die Original-Routine des Betriebssystems. Es spielt auch keine Rolle, ob das zu vergleichende Programm teilweise oder ganz unterm ROM liegt. Bei der Fehlersuche erzeugt das Programm eine Tabelle auf dem Bildschirm, in der die fehlerhaften Adressen und deren Byte-Inhalt als Dezimalzahlen angezeigt werden. Am Ende seiner Nachforschungen bringt der Computer als letzte Meldung Start- und Endadresse des durchforsteten Programms.

Laden Sie das Utility mit:

```
LOAD "VERIFY-MASTER V1",8,8
```

Die Anweisung VERIFY gilt jetzt nicht mehr. Der neue Vergleichsbefehl lautet:

```
SYS 49152, "(Programmname)"
```

Der Bildschirm wird in drei Spalten eingeteilt: Fehleradresse (Programm im Speicher), Bytewert (Programm im Speicher), Filewert (Version auf Diskette).

»Verify-Master« belegt nach dem Laden den Speicherbereich von $C000 bis $C1DF. Das dokumentierte Quellisting »Verify-Quellcode« läßt sich mit »Hypra-Ass« laden und ändern. Zunächst öffnet das Programm unter gleichzeitiger Ausgabe des Arbeitsbildschirms die zu vergleichende Datei auf Diskette ($C000 bis $C030), liest die Startdadresse und vergleicht sie nun Byte für Byte mit dem Programm im Basic-RAM ($C031 bis $C05A). Falls Unterschiede auftauchen, verzweigt das Programm zur entsprechenden Routine, die für die Fehlerliste zuständig ist ($C05B bis $C092).

(Matthias Strecker/bl)

FRAGEN & ANTWORTEN

Täglich erreichen uns Briefe, gespickt mit Fragen zur Assembler-Programmierung. Vielleicht finden Sie auf den folgenden Seiten auch die Lösung Ihres Problems.

Multitasking

Wie kann ich mehrere Maschinenprogramme gleichzeitig ablaufen lassen?

Unser Vorschlag: Verbiegen Sie die IRQ-Vektoren $0314/0315 und richten Sie diese auf Ihre eigene Maschinen-Routine. Als Kennwert können Sie z.B. die Bildschirm-Rasterzeile abfragen. Hat sie einen bestimmten Punkt erreicht, muß der Interrupt-Vektor auf das andere Maschinensprache-Programm gesetzt werden.

Ein Beispiel:

```
SETIRQ   sei
         ldx #$<IRQ1
         ldy #$>IRQ1
         stx $0314
         sty $0315
         cli
         rts
IRQ1     lda $d019
         sta $d019
         lda #$00
         sta $d020
         sta $d021
         lda #$50
         sta $d012
         sei
         lda #$<IRQ2
         sta $0314
         cli
         jmp $febc
;holt x-, y-Register und
;Akku vom Stapel
IRQ2     lda $d019
         sta $d019
         lda #$01
         sta $d020
         sta $d021
         lda #$00
         sta $d012
         sei
         lda #$<IRQ1
         sta $0314
         cli
         jmp $ea31
```

Dieses simple Beispiel bedient sich zweier Maschinenspracheprogramme: IRQ1 stellt die Rahmen- und Bildschirmfarbe schwarz ein, IRQ2 setzt die Farben auf Weiß. Beachten Sie, daß beide Routinen in einem Block beginnen. Unser Beispielprogramm »Multitask« befindet sich auf der Diskette zu diesem Sonderheft. Sie laden es mit:

`LOAD "MULTITASK",8,1`

und starten mit »SYS 49152«.

Bei dieser kurzen Routine wird eines deutlich: Multitasking verspricht mehr, als es hält: Obwohl es so aussieht, werden sämtliche Programmschritte hintereinander und einzeln abgearbeitet. Lediglich durch die immense Geschwindigkeit von Maschinensprache sieht es so aus, als würden die Programme **gleichzeitig** ablaufen.

Funktionstasten belegen

Wie kann ich unter Maschinensprache die Funktionstasten des C64 mit Text belegen?

Im Interrupt (IRQ) fragt der C64 neben anderen Aufgaben auch nach der aktuell gedrückten Taste. Dies gilt selbstverständlich auch für die Funktionstasten <F1> bis <F8>. Dazu muß man die IRQ-Vektoren $0314/$0315 auf eine eigene Routine verbiegen, die per Druck auf eine Funktionstaste den gewünschten Text ausgibt. Anschließend macht das Programm in der normalen IRQ-Routine ($EA31) weiter.

Unser Beispielprogramm überprüft z.B. die Tasten <F1>, <F3>, <F5> und <F7>. Sie sind mit folgenden Texten belegt, die im Bereich ab Label FTEXT gespeichert sind:

- <F1>: SAVE,
- <F3>: LIST + <RETURN>,
- <F5>: RUN + <RETURN>,
- <F7>: LOAD

Hier das Assembler-Programm mit Erläuterung:

```
; IRQ-Vektor auf eigene
; Routine richten:
SETIRQ  sei
        ldx #$<FTAST
        ldy #$>FTAST
        stx $0314
        sty $0315
        cli
        rts
; Original-IRQ-Routine
; wieder einschalten:
IRQNORM sei
        ldx #$31
        ldy #$ea
        stx $0314
        sty $0315
        cli
        rts
; eigene Tastenabfrage:
FTAST   lda $c5
; Codewerte: <F1> = 4,
; <F3> = 5, <F5> = 6,
; <F7> = 3. Bei jeder
; größeren Zahl als >>3<<
; bleibt das Carry-Flag
; gelöscht:
        cmp #$03
        bcc $STATEMP
; Wert größer als >>6<<?
; Dann ist Carry-Flag aktiv!
        cmp #$07
        bcs $STATEMP
; vergleiche mit Adresse
; TEMP, die den aktuellen
; Tastencode speichert:
        cmp $TEMP
        beq $STATEMP
F1      ldy #$00
        ldx #$03
 CODE   cpx $c5
; wenn <F7>, verzweige:
        beq $F7
; y-Zähler erhöhen:
YINC    iny
; Text für <F1> laden:
        lda $FTEXT,y
; noch kein Null-Byte
; aufgetaucht? Dann
; weitermachen:
        bne $YINC
; x-Reg. erhöhen:
        inx
; größer als >>0<<? Dann
; vergleiche Wert mit Inhalt
; von Adresse 197:
        bne $CODE
F7      ldx #$00
; erhöhe x- und y-Register:
 XINC   inx
        iny
; hole Text:
        lda $FTEXT,y
; Tastaturpuffer hochzählen:
        sta $0276,x
; fertig, wenn $00 gelesen
; wurde:
        bne $XINC
; Anzahl im Tastaturpuffer:
        stx $c6
; aktuellen Tastencode in
; Zwischenspeicher:
        lda $c5
STATEMP sta $TEMP
; zurück zur IRQ-Routine:
        jmp $ea31
TEMP $00
FTEXT (ab hier stehen die
Textbytes als ASCII-Codes
($00 dient als Endekennung).
```

Entsprechende Label-Bezeichnungen sind großgeschrieben. Wer sich Tipparbeit sparen möchte, lädt das Programm von der Diskette zum Sonderheft:

`LOAD "KEYS",8,1`

Es liegt im Speicherbereich ab $C000 und wird mit »SYS 49152« gestartet. Die Funktionstasten <F1>, <F3>, <F5> und <F7> sind nun mit Texten belegt, die nach Tastendruck auf dem Bildschirm erscheinen. Mit »SYS 49165« kann man das Utility abschalten: Die IRQ-Vektoren erhalten wieder ihre Originalwerte.

Teepause ohne Hast

Gibt's eine Tastenkombination im C64 (wie z.B. <CTRL S> beim PC), mit der man ein laufendes Programm oder die Ausgabe eines Listings beliebig lang anhalten kann? Die Funktion der CTRL-Taste (verlangsamte Bildschirmausgabe) stellt mich nicht zufrieden.

Der C64 kennt im Betriebssystem keine Routine, die diese Pausenfunktion erfüllt: Die muß man sich selbst programmieren! Dazu klinkt man sich in die Tastaturabfrage ein, die jede 1/60-Sekunde im Interrupt durchgeführt wird.

Unser Beispielprogramm beobachtet die Leertaste <SPACE>. Wird sie gedrückt, fungiert sie als Pausentaste - der Computer unterbricht seine Arbeit (z.B. bei der Listing-

Ausgabe oder in einem Basic-Programm). Tippt man erneut auf <SPACE>, geht's wieder normal weiter.

Das entsprechende Assembler-Listing:

```
;IRQ-Vektor verbiegen:
SETIRQ     sei
           ldx #$<PAUSE
           ldy #$>PAUSE
           stx $0314
           sty $0315
           cli
           rts
; aktuell gedrückte Taste
; abfragen (Adresse 203):
PAUSE      lda $cb
; ist es der Codewert 60 für
; die Leertaste?
           cmp #$3c
; nein, dann weiter im
; normalen Interupt:
           bne $IRQNORM
; ja, dann springe zur
; Tastaturabfrage im
; Betriebssystem:
ASKKEY     jsr $ea87
; Tastaturpuffer mit dem
; Wert >>1<< belegt?
           lda $c6
           cmp #$01
; ja, dann zurück zur
; Systemroutine!
           beq $ASKKEY
; nein, dann Tastaturpuffer
; löschen und weiter im
; Interrupt:
IRQNORM    lda #$00
           sta $c6
           jmp $ea31
```

Der Trick funktioniert, weil nach <SPACE> der Wert »1« in der zuständigen Speicherstelle für den Tastaturpuffer (Adresse 198) gespeichert wird. Der Computer befindet sich dann so lange in einer Warteschleife, bis erneut eine Taste gedrückt wird: Die Zahl im Tastaturpuffer ist dann ungleich (höher als) »1«. Wenn diese Bedingung erfüllt ist, macht der C 64 wie gewohnt weiter. Dieses Beispielprogramm finden Sie auf der Diskette zu diesem Sonderheft. Es belegt den Speicherbereich ab $C000.

Laden Sie es mit:

```
LOAD "PAUSE",8,1
```

Geben Sie NEW ein und starten Sie die Pausenfunktion mit »SYS 49152«.

Betriebssystem kopieren

Ich suche eine Maschinenroutine, die bei $9000 liegen soll und den Basic-Interpreter vom ROM ins darunterliegende RAM kopiert.

Diesen Wunsch erfüllt folgendes Assembler-Programm:

```
; Anfangsadresse des
; Bereichs ($A000), der
; kopiert werden soll
; (Low- und Highbyte) in die
; Adressen 95 und 96:
ROMCOPY lda #$00
        ldy #$a0
        sta $5f
        sty $60
; Endadresse + 1 in die
; Speicherstellen 90/91 und
; 88/89:
        lda #$00
        ldy #$c0
        sta $5a
        sty $5b
; wichtig: zweimal!
        lda #$00
        ldy #$c0
        sta $5a
        sty $5b
        lda #$00
        ldy #$c0
        sta $58
        sty $59
; Blockverschiebe-
; Routine des Betriebs-
; Systems:
        jsr $a3bf
        rts
```

Laden Sie das Programm mit:

```
LOAD "ROM-COPY",8,1
```

Gestartet wird es mit »SYS 36864«. Um aber das RAM unterm ROM zu erreichen, muß man Bit Nr. 0 in Adresse 1 löschen:

```
POKE 1,PEEK(1) AND 254
```

Der Normalzustand wird mit folgender Anweisung wiederhergestellt:

```
POKE 1,PEEK(1) OR 1
```

Dem Computer ist es egal, welche Konfiguration aktiv ist, denn eine originalgetreute Kopie des Basic-Interpreters steht jetzt auch im RAM zur Verfügung. Andernfalls würde er abstürzen.

Diese Kopierroutine ist universell. Es lassen sich damit beliebige Speicherbereiche verschieben (z.B. Hires-Grafiken usw.). Wichtig: Die Anfangsadresse des Bereichs kommt in die Adresse $5F/$60, die Endadresse + 1 muß in den Speicherstellen $58/$59 und $5A/$5B stehen. Durch eine Besonderheit des Bereichs für Fließkomma-Operationen müssen die Adressen $5A/$5B zweimal beschrieben werden.

Geteilter Bildschirm

Für ein selbstprogrammiertes Adventure möchte ich den Bildschirm teilen. In der oberen Hälfte soll Hires-Grafik, darunter der Text gezeigt werden. Gibt es entsprechende POKEs?

Mit POKEs allein ist es nicht getan. Man muß einen Split-Screen erzeugen: Hires- und Textmodus auf ein und demselben Bildschirm. Wichtig ist die Speicherstelle $D012. Darin ist stets die aktuelle Nummer der Rasterzeile gespeichert, die der Kathodenstrahl des Monitors beim ständigen Aufbau des Bildschirms durchläuft. Da der Rasterzeilenwert »255« übersteigen kann, dient das Bit 7 von Adresse $D011 als Übertrags-Flag. Wenn man jetzt einen bestimmten Wert in Adresse $D012 einträgt, wird diese Zahl zwischengespeichert und dauernd mit dem aktuellen Rasterzeilenwert verglichen. Stimmen die Inhalte überein, findet der Raster-Interrupt statt: Bit 0 in Adresse $D019 wird gesetzt. Soll gleichzeitig ein Interrupt des Mikroprozessors stattfinden, muß zusätzlich Bit 0 der Speicherstelle $D01A eingeschaltet werden. Der IRQ-Vektor in $0314/0315 wird nun auf eine Maschinensprache-Routine gerichtet, die den geteilten Bildschirm initialisiert. Unser Assembler-Listing sieht so aus:

```
; IRQ-Vektoren umstellen:
SETIRQ  ldx #$<SPLIT
        ldy #$>SPLIT
        stx $0314
        sty $0315
; Bit 7 in $D01A
; einschalten:
        lda #$81
        sta $d01a
        cli
        rts
; Inhalt von $D019
; prüfen, ob Inhalt = 1:
SPLIT   lda $d019
        and #$01
; ja, dann weiter im
; normalen Interrupt:
        beq $IRQNORM
; $D019 mit neuem Wert
; beschreiben:
        sta $d019
        lda #$3B
; auf Stapel retten:
        pha
        lda #$18
        pha
; Zeile, an der der Screen
; geteilt wird (208):
        ldx #$d0
        lda $d012
        bpl $GRAFIK
; Werte vom Stapel holen:
        pla
        pla
        lda #$1b
        pha
        lda #$15
        pha
        ldx #$10
GRAFIK  stx $d012
        pla
; Grafik-Modus ein/aus:
        sta $d018
        pla
        sta $d011
        jmp $febc
IRQNORM jmp $ea31
```

Laden Sie das Programm mit:

```
LOAD "SPLITSCREEN",8,1
```

Da es ebenfalls im Bereich von $C000 liegt, startet man es mit »SYS 49152«.

Auf dem Bildschirm sind beide Modi sichtbar: Bis Rasterzeile 208 findet man Hires-Grafik (die allerdings noch nicht gelöscht ist), darunter fünf Textzeilen. Es ist möglich, daß sich der Text-Cursor unter dem Hires-Bereich versteckt: Mit den Cursor-Tasten läßt er sich an die gewünschte Stelle im unteren Bildbereich dirigieren. Wer die Trennposition ändern möchte, die den Raster-Interrupt auslöst, muß unter Adresse $C023 im Assembler-Listing einen anderen Wert eintragen.

Laden und Speichern in Assembler

Lassen sich die Floppy-Befehle LOAD und SAVE auch in einem Maschinenprogramm aufrufen?

Ja, dazu gibt es Routinen im Betriebssystem, die allerdings erst vorbereitet werden müssen: **LOAD ($FFD5)** und **SAVE ($FFD8).**

LOAD ($FFD5): Das x-Register muß das Low-Byte, das y-Register das High-Byte der Anfangsadresse enthalten. Der

SONDER HEFT 72 VORSCHAU

Was Sie schon immer über Drucker wissen wollten
finden Sie u.a. in einem leicht verständlichen Kurs. Außerdem bekommen Sie alles Know-how zur reibungslosen Funktion Ihres Druckers.

■ Welcher Drucker hat welche Vorzüge und Schwächen? Diese und andere Fragen beantworten wir auf fünf Seiten Druckertest.

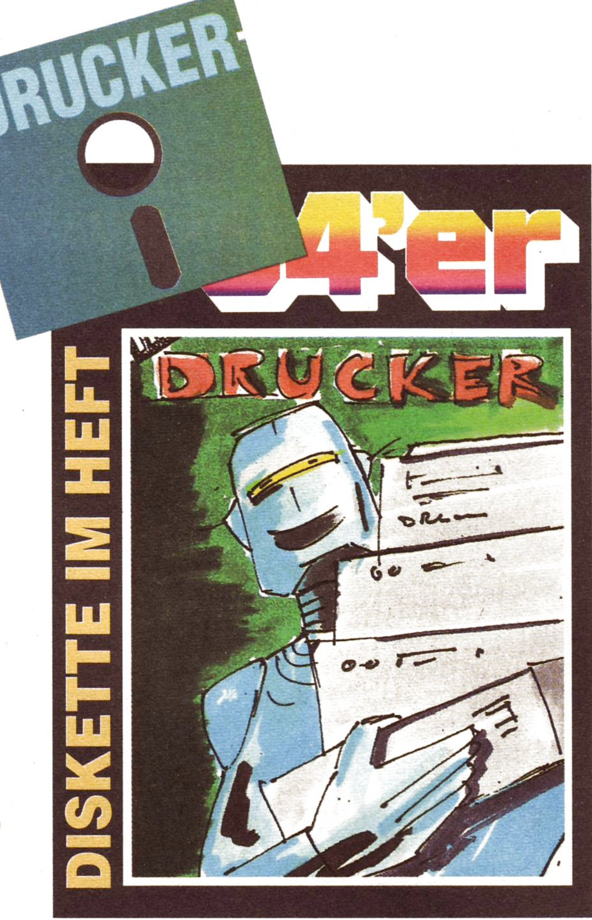
■ »Publish 64« macht Sie zum Redakteur. Mit diesem DTP-Programm erzeugen Sie alles schwarz auf weiß: Von der Vereinszeitschrift bis zur Illustrierten.

■ Eigene Briefköpfe, Schilder, Glückwunschkarten und Spruchbänder zu entwerfen, sind nur einige der Funktionen, die Ihnen »Topprint« zur Verfügung stellt. Es dient auch als intelligentes Hardcopy-Programm.

■ Wenn Sie Ihren neuen Drucker nun endlich daheim haben, zeigen wir Ihnen, wie man ihn am C 64 zum Arbeiten bringt. Wir verraten Kniffe, die nicht in jedem Handbuch stehen.

■ Eine geballte Ansammlung von Tips und Tricks machen den Umgang mit dem Drucker zum Vergnügen.

■ Mit »Fontprint« haben Sie endlich die Möglichkeit, geänderte Zeichensätze nicht nur am Bildschirm zu bewundern.



Das Sonderheft 72 finden Sie ab 22.11.1991 bei Ihrem Zeitschriftenhändler.

Aus aktuellen oder technischen Gründen können Themen verschoben werden. Wir bitten um Ihr Verständnis.

Akku dient als Zeiger, ob VERIFY (Akku = 1) oder LOAD (Akku = 0) stattfinden soll. Selbstverständlich muß man auch bei dieser Routine zunächst die File-Parameter und den Dateinamen setzen. Ist die Sekundäradresse »0«, wird das Programm wie ein Basic-Programm geladen (an $0801), bei »1« absolut, also wie ein Maschinenprogramm. Als Beispiel soll eine Hires-Grafik nach $2000 geladen werden:

```
ldy #$00 ;Sekundäradresse
ldx #$08 ;Geräteadresse
jsr $ffba ;Dateiparameter
ldx #$<FILENAME
ldy #$>FILENAME
lda #$LAENGE
jsr $ffbd
ldx #$00 ;Lowbyte von $2000
ldy #$20 ;Highbyte
lda #$00 ; Modus 0 = laden
jsr $ffd5
rts
FILENAME .by 4e 41 4d 4e
LAENGE .by $03
```

Dieses Beispielprogramm finden Sie auf unserer Diskette zum Sonderheft. Es befindet sich im Speicherbereich ab $C000. Wenn Sie die Routine für eigene Programme verwenden möchten, sollten die Bytes des Dateinamens ab $C01A und unmittelbar dahinter die Angabe der Namenslänge stehen. Beachten Sie, daß der Computer immer ab »0« zählt.

SAVE ($FFD8): Die Parameterverteilung ist hier anders als bei LOAD: x- und y-Register enthalten Low- und High-Byte der **Endadresse + 1** des zu speichernden Bereichs. Die Startadresse befindet sich jetzt in zwei aufeinanderfolgenden Speicherstellen in der Zeropage (z.B. $FB/$FC). Die Dateiparameter und der Filename müssen ebenfalls vorher gesetzt werden. Bei SAVE besitzt die Sekundäradresse folgende Bedeutungen: 0 = Basic-Programm, 1 = Maschinensprache-Programm.

Unser Beispiel speichert eine Hires-Grafik ab $2000 auf Diskette:

```
ldy #$01 ;Sekundäradresse
ldx #$08 ;Gerätenummer
jsr $ffba
ldx #$<FILENAME
ldy #$>FILENAME
lda #$LAENGE
jsr $ffbd
ldx #$40 ;Lowbyte Ende+1
ldy #$40 ;Highbyte Ende
lda #$fb ;Startadresse
jsr $ffd8
rts
FILENAME .by 4e 41 4d 45
LAENGE .by $04
```

Bevor man diese Routine von Diskette lädt und startet, müssen Dateiname und Länge ebenfalls ab Adresse $C01A stehen. Außerdem sollten Sie die Anfangsadresse $2000 in $FB/$FC (251/252) als Low- und High-Byte eintragen.